TAMPERE UNIVERSITY OF TECHNOLOGY

Anum Imran

SOFTWARE IMPLEMENTATION AND PERFORMANCE OF UMTS TURBO CODE

Masters of Science Thesis

Examiners: Professor Jari Nurmi
Professor Mikko Valkama
Examiner and topic approved in the Faculty of Computing and Electrical Engineering Council meeting on 15 August, 2012.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Program in Information Technology
Faculty of Computing and Electrical Engineering
Imran, Anum: The UMTS turbo code implementation.
Master of Science Thesis: 74 pages, 22 appendix pages.
March 2013
Major: Communication Engineering
Examiners: Prof. Jari Nurmi and Prof. Mikko Valkama.
Keywords: Coding, turbo codes, UMTS, MATLAB simulation.

In the recent years, there has been a proliferation of wireless standards in television, radio and mobile communications. As a result, compatibility issues have emerged in wireless networks. The size, cost and competitiveness set limitations on implementing systems compatible with multiple standards. This has motivated the concept of software defined radio which can support different standards by reloading the software and implementing computationally intensive parts on hardware, e.g., iterative codes. In a typical communication system, all the processing is done in the digital domain. The information is represented as a sequence of bits which is modulated on an analog waveform and transmitted over the communication channel. Due to channel induced impairments, the received signal may not be a true replica of the transmitted signal. Thus, some error control is required which is achieved by the use of channel coding schemes that protect the signal from the effects of channel and help to reduce the bit error rate (BER) and improve reliability of information transmission.

Shannon gave the theoretical upper bound on the channel capacity for a given bandwidth, data rate and signal-to-noise ratio in 1940s but practical codes were unable to operate even close to the theoretical bound. Turbo codes were introduced in 1993 where a scheme was described that was able to operate very close to the Shannon limit. Turbo codes are widely used in latest wireless standards e.g. UMTS and LTE. A basic turbo encoder consists of two or more component encoders concatenated in parallel and separated by an interleaver. The turbo decoder uses soft decision to decode the bits and the decoding is done in an iterative fashion to increase reliability of the decision.

In this thesis, the turbo code for the UMTS standard is implemented in MATLAB. Four versions of the Maximum Aposteriori Probability (MAP) algorithm are used in the implementation. The simulation results show that the performance of the turbo code improves by increasing the number of iterations. Also, better performance can be achieved by increasing the frame size or the interleaver size and increasing the signal power. Overall, the designing of turbo codes is a trade-off between energy efficiency, bandwidth efficiency, complexity and error performance.

PREFACE

I would like to thank my supervisors, Professor Jari Nurmi and Professor Mikko Valkama, for giving me the opportunity to research on this interesting topic for my master's thesis. I wish to offer special thanks to Professor Jari Nurmi for his valuable feedback, comments and encouragement during the course of the thesis.

I would also like to acknowledge the ideal and productive research environment provided by Tampere University of Technology to work on my thesis.

In the end, I would like to thank my husband, my parents and especially my son for being helpful, patient and supportive during the whole period of my studies.

Tampere, February 24, 2013.

Anum Imran

LIST OF ABBREVIATIONS

A/D	Analog-to-Digital
ADC	Analog-to-Digital Converter
ACS	Add Compare Select
APP	A Posteriori Probability
ASIC	Application Specific Integrated Circuit
ATSC	Advanced Television Systems Committee
AWGN	Additive White Gaussian Noise
BCH	Bose Chaudhuri Hocquenghem
BER	Bit Error Rate
BPSK	Binary phase Shift keying
CML	Coded Modulation Library
D/A	Digital-to-Analog
DAC	Digital-to-Analog Converter
DAB	Digital Audio Broadcasting
DCS	Digital Communication System
DSP	Digital Signal Processor
DVB	Digital Video Broadcasting
FPGA	Field Programmable Gate Array
gcd	Greatest Common Divisor
GSM	Global System for Mobile Communication
IF	Intermediate Frequency
LLR	Log Likelihood Ratio
LTE	Long Term Evolution

MAP	Maximum A Posteriori
MLD	Maximum Likelihood Decoder
OTA	Over the Air
PCCC	Parallel Concatenated Convolutional Code
PCM	Pulse Code Modulation
pdf	Probability density function
RF	Radio Frequency
RS Code	Reed Solomon Code
RSC	Recursive systematic Convolutional
SDR	Software Defined Radio
SISO	Single Input Single Output
SoC	System on Chip
SOVA	Soft Output Viterbi Algorithm
UMTS	Universal Mobile Telecommunication System
WLAN	Wireless Local Area Network

LIST OF FIGURES

LIST OF TABLES

Table of Contents

1. INTRODUCTION

In the recent years, there has been a proliferation of wireless standards in television, radio and mobile communications. As a result, compatibility issues have emerged in wireless networks. Some of the most popular standards include wireless local area network (WLAN), i.e., IEEE 802.11; 2.5G/3G mobile communications, i.e., Global System for Mobile Communication (GSM), Universal Mobile Telecommunication System (UMTS), Long Term Evolution (LTE); digital television standards, i.e., Digital Video Broadcast (DVB), Advanced Television Systems Committee (ATSC) standards; digital radio standards, i.e., Digital Audio Broadcasting (DAB). The inconsistency between the wireless standards is causing a lot of problems to equipment vendors, network operators and subscribers. Equipment vendors face difficulties in airing new technologies because of short time-to-market requirements. The subscribers are forced to change their handsets to upgrade themselves to the new standards whereas the network operators face the dilemma during upgrade of network from one generation to another due to large number of subscribers with handsets incompatible with the new generation of standards [1].

The design of highly flexible digital communication has become an area of great interest in the recent years as the inconsistencies between wireless standards is inhibiting deployment of global roaming facilities and causing problems in introducing new features and services. Also, there are demands of improved services and cheaper rates from customers. The rapid evolution of communication technology is pushing the service providers to keep pace with latest technology trends to survive in the market. Traditional wireless systems with hard-coded capabilities are no longer able to keep step with the rapid growth rate of communication technologies. Such devices, e.g. cell phones and two way radios, etc, with fixed embedded software have significant limitations and they become obsolete with the introduction of new services and technologies in market, thus requiring expensive upgrades of total replacement. Software defined radio (SDR) is one way to address these issues.

1.1 Software defined radio

In mobile wireless transmissions, size, cost and competitiveness set limitations on implementing systems compatible with multiple standards. This has motivated the concept of software defined radio which can support different standards by reloading the software. SDR term refers to reconfigurability and adaptability of radio modules using software. It is basically a collection of hardware and software technologies where

the radio functionalities are implemented as software modules running on generic hardware platforms such as field programmable gate arrays (FPGA), Digital Signal Processors (DSP) and Programmable System on Chip (SoC). Multiple software modules implementing different standards can be present in the radio system. The radio system can take up different personalities depending on the software module being used [2].

The SDR technology facilitates implementation of future-compliant systems. All the concerned parties, i.e. subscriber, network operator and handset manufacturer, benefit from this scheme. Both network infrastructure and subscriber handsets can be implemented based on SDR concept. It makes the system very flexible and allows easy migration of networks from one generation to another. In the same way subscriber handset can adapt to various network protocols by choosing the suitable software module. New software modules can be transferred to the subscriber handset using over the air (OTA) upload [3]. Network operator can roll out new services at a much faster pace by mass customization of subscriber handsets. Manufacturers can improve the quality of their products and remove bugs by software upgrades.

SDR is thus a technology for building systems which support multiple services, multiple standards, multiple bands, multiple modes and offer diverse services to its user. Functional modules in a radio system such as modulator/demodulator, signal generator, channel coding, multiplexing and link-layer protocols can be implemented based on SDR concept. This helps in building reconfigurable software radio systems where dynamic selection of parameters for each of the above-mentioned functional modules is possible [4] [5].

1.2 Architecture of SDR

The digital radio system is composed of three main functional blocks: Radio Frequency (RF) section, Intermediate Frequency (IF) section and baseband section [1]. To achieve the flexibility and adaptability required by SDR, the boundary of digital processing should be moved as close as possible to the antenna. Thus for an SDR system, the analog-to-digital (A/C) and digital-to-analog (D/A) wideband conversion is done in the IF section instead of baseband section as done for conventional radio. Figure 1.1 shows the block diagram of a digital radio system [1].

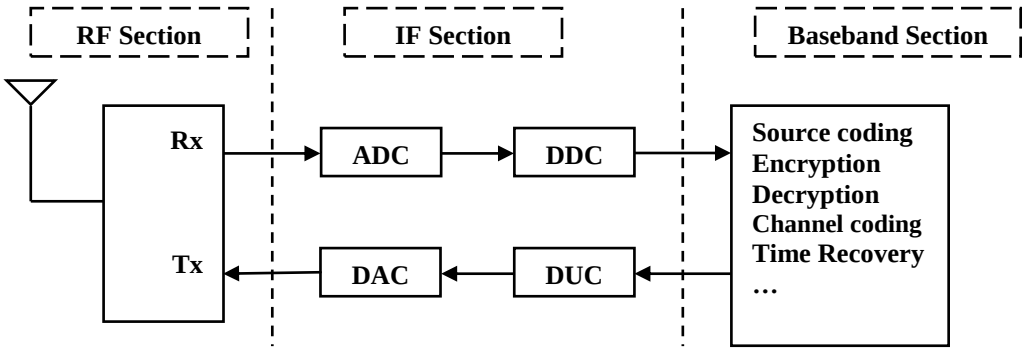


Figure 1.1 Block diagram of a digital radio system.

1.2.1 The RF section

The RF section is responsible for transmitting and receiving the RF signal and converting the RF signal into an IF signal. The RF section consists of antennas and analog hardware modules. The RF front-end on the receive side performs RF amplification and analog-down-conversion (from RF to IF). On the transmit side, the RF section performs analog-up-conversion (from IF to RF) and RF power amplification. The RF front-end is designed in such a way to reduce interference, multipath and noise.

1.2.2 The IF section

The IF section performs two tasks in either direction. On the receiver path, the analog-to-digital converter (ADC) functional block performs A/D conversion followed by Digital-down-conversion (Demodulation) by the DDC functional block. On the transmit path, the D/A conversion is done by digital to analog converter (DAC) functional block and Digital-up-conversion (modulation) by DUC functional block. Digital filtering and sample rate conversion are often needed to interface the output of the ADC and DAC to the processing hardware at the receiver and transmitter respectively.

1.2.3 The baseband section

In the baseband section baseband operations like channel coding, source coding, equalization, encryption, decryption, modulation, demodulation, frequency hopping and timing recovery are carried out. To allow reconfigurability and flexibility, application-specific integrated circuits (ASICs) are replaced with reprogrammable or reconfigurable modules in software defined radio [3]. In this thesis, the main area of focus is the baseband section in which all the major tasks of a basic digital communication system (DCS) are performed.

1.3 Digital communication system

Recent progress in DCS design rests mainly upon software algorithms instead of dedicated hardware. In a typical communication system, all the processing is done in the digital domain. Digital transmission offers data processing options and flexibilities not available with analog transmission. The principle feature of a DCS is that during a finite interval of time, it sends a waveform from a finite set of possible waveforms whereas for an analog communication system, the waveform can take any shape. The objective of the receiver is to determine which waveform from a finite set of waveforms was transmitted. A typical DCS illustrating signal flow and the signal processing steps is shown in Figure 1.2 [6].

The upper blocks denote signal transformations at the transmitter end whereas the lower blocks indicate the receiver end. The information source inputs analog information into the system which is converted into bits, thus assuring compatibility between the information source and signal processing within the DCS. Source coding is then done to remove redundant information by quantization and compression. Encryption is done to prevent unauthorized users from understanding messages and injecting false messages and thus ensure data privacy and integrity. Next channel coding is performed by adding redundant bits to the data for error detection and correction. It increases the reliability of data at the expense of transmission bandwidth or decoder complexity. Multiplexing and multiple access procedure combine signals that might have different characteristics of different sources so that they can share the same communication source.

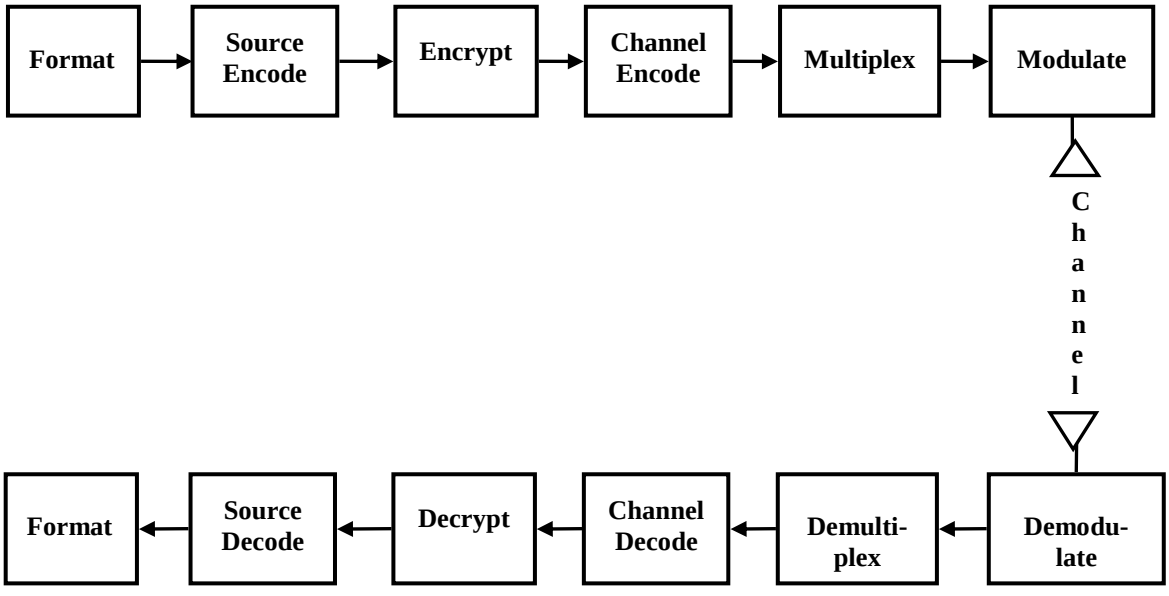


Figure 1.2 Block diagram of a digital communication system (adapted from [6]).

The pulse modulate block usually includes filtering to minimize the transmission bandwidth. It converts the binary symbols to a pulse-code-modulation (PCM) waveform. For applications involving RF transmission, the next important step is band pass modulation which translates the baseband waveform to a much higher frequency using carrier wave. Frequency spreading produces a signal that is invulnerable to interference and noise thus enhancing privacy. The signal processing steps that take place in the transmitter are reversed in the receiver.

This thesis focuses on channel coding for SDR. The thesis gives an overview of different type of channel codes and then narrows the research work to Turbo Codes. Turbo Codes are widely used in modern communication systems. In this thesis, Turbo Codes are implemented for UMTS system and their performance is analysed under various channel conditions, frame sizes, signal power and iterations.

1.4 Thesis overview

The first part of the thesis deals with the study of basic turbo codes. A detailed study of the encoder, the encoding algorithm, interleaver design and MAP decoding algorithm has been done. Then the UMTS turbo code implementation has been discussed.

The thesis is organized into six chapters. Chapter 1 discusses the essence of Software Defined Radio with emphasis on baseband module. The elements of the basic digital communication are discussed. Chapter 2 discusses the basic channel coding schemes and their importance. The discussion includes an overview of the performance of various channel codes and the need of better channel coding schemes. Turbo codes are discussed in detail in Chapter 3 including the encoder and decoder architecture, the interleaver design and decoding algorithms. Chapter 4 concentrates on the turbo code for UMTS standard. The specifications of the UMTS turbo encoder, channel and decoder are discussed. The MATLAB implementation details and the simulation results are given in Chapter 5. Chapter 6 concludes and summarizes the thesis.

2. CHANNEL CODING

Over the years, there has been a tremendous increase in the trends of digital communication especially in the fields of cellular, satellite and computer communications. In the digital communication system, the information is represented as a sequence of bits. The processing is done in the digital domain. The binary data is then modulated on an analog waveform and transmitted over the communication channel. The signal is corrupted by the noise and interference introduced by the communication channel. At the receiver end, the corrupted signal is demodulated and mapped back to the binary bits. Due to channel induced impairments, the received signal may not be a true replica of the transmitted signal; rather it is just an estimate of the transmitted binary information. The bit error rate (BER) of the received signal depends on the noise and interference of the communication channel. In a digital transmission system, error control is achieved by the use of channel coding schemes. Channel coding schemes protect the signal from the effects of channel noise and interference and ensure that the received information is as close as possible to the transmitted information. They help to reduce the BER and improve reliability of information transmission.

2.1 Introduction

Channel coding schemes involve the insertion of redundant bits into the data stream that help to detect and correct bit errors in the received data stream. Due to the addition of the redundant bits, there is a decrease in data rate. Thus the price paid for using channel coding to reduce bit error rate is a reduction in data rate or an expansion in bandwidth.

2.2 Types of channel codes

There are two main types of channel codes, block codes and convolutional codes [7]. Block codes accept a block of *k* information bits, perform finite field arithmetic or complex algebra, and produce a block of n code bits. These codes are represented as *(n, k)* codes. The encoder for a block code is memory less, which means that the n digits in each codeword depend only on each other and are independent of any information contained in previous codeword. Some of the common block codes are Hamming codes and Reed Solomon (RS) codes. RS codes are non-binary cyclic error correcting codes that could detect and correct multiple random symbol errors. A Hamming code is a linear error-correcting code which can detect up to two simultaneous bit errors, and correct single-bit errors. For multiple error corrections, a generalization of Hamming codes known as Bose Chaudhuri Hocquenghem (BCH) codes is used.

Block codes can either detect or correct errors. On the other hand, convolutional codes are designed for real-time error correction. The code converts the entire input stream into one single codeword. The encoded bit depends not only on the current bit but also on the previous bit information. The decoding is traditionally done using the Viterbi algorithm [8].

2.3 Need for better codes

The design of a channel code is always a trade-off between energy efficiency and bandwidth efficiency [9]. Low rate codes having more redundant bits can usually correct more errors. That means that the communication system can operate at lower transmit power, tolerate more interference and noise and transmit at higher data rate. Thus the code becomes more energy efficient. However, low rate codes also have a large overhead and have more bandwidth consumption. Also, the decoding complexity of the code also grows exponentially with code length. Thus, low rate codes set high computational requirements to the conventional decoders.

There is a theoretical upper limit on the data transmission rate for a given bandwidth, channel type, signal power and received noise power such that the data transmission is error-free. The limit is called the channel capacity or the Shannon capacity. The formula for additive white Gaussian noise (AWGN) channel is

$$R < W log_2 \left(1 + \frac{S}{N}\right) bits/sec \tag{2.1}$$

Where W is the bandwidth, S is signal power, N is received noise power and R is the data transmission rate. In practical transmission, no such thing as an ideal error free channel exists. Instead, the bit error rate is brought to an arbitrarily small constant often chosen at 10^{-5} or 10^{-6}. Shannon capacity sets a limit on the energy efficiency of the code as it defines the lower bound for the amount of energy that be expended to convey one bit of information given a fixed transmission rate, bandwidth and noise power.

Shannon gave his theory in 1940s but practical codes were unable to operate even close to the theoretical bound. Until the beginning of 1990s, the gap between these theoretical bounds and practical implementations was still at best about 3dB, i.e., the practical codes required about twice as much energy as the theoretical predicted minimum. Efforts were made to discover new codes that allow easier decoding. That led to the introduction of concatenated codes. Simple codes were combined in parallel fashion so that each part of the code can be decoded separately, thus decreasing decoder complexity. Also, the decoders can exchange information with each other to increase reliability. This lead to the introduction of near Shannon capacity error correcting code known as turbo code.

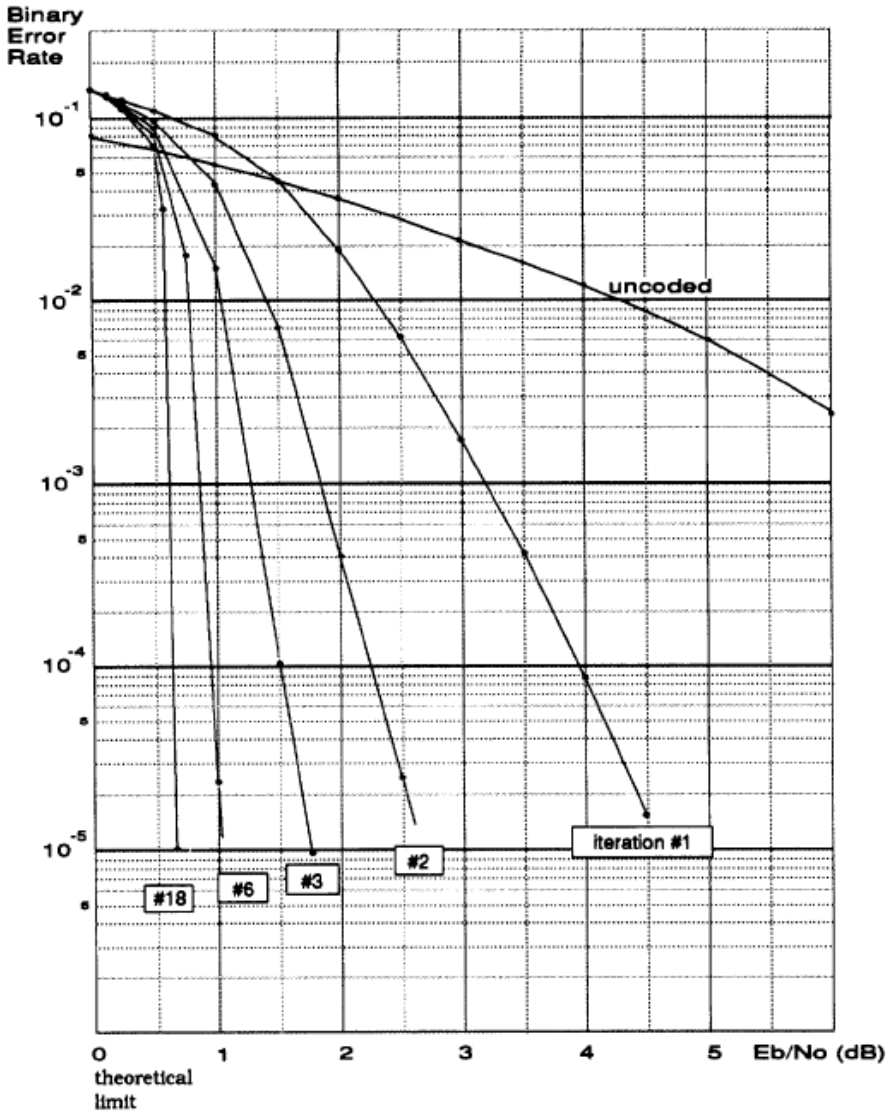


Figure 2.1 Bit error rate for iterations (1,2…18) of the rate ½ turbo code presented in 1993 [10] .

The first turbo code, based on convolutional encoding, was introduced in 1993 where a scheme was described that uses a rate ½ code over an AWGN channel and achieves a bit error probability of 10^{-5} using Binary Phase Shift Keying (BPSK) modulation at an E_b/N_o of 0.7 dB [9]. The BER curves for different iterations of rate r=1/2 turbo code are shown in Figure 2.1 [10].

The turbo codes consist of two or more component encoders separated by interleavers so that each encoder uses an interleaved version of the same information sequence. Contrary to conventional decoders which use hard decision to decode the bits, concatenated codes like turbo codes do not limit themselves by passing hard decision among the decoders. They rather exchange soft decision from the output of one decoder to the input of the other decoder. The decoding is done in an iterative fashion to increase reliability of the decision.

3. TURBO CODES

The generic form of a turbo encoder consists of two encoders separated by the interleaver. The two encoders used are normally identical and the code is systematic, i.e., the output contains the input bits as well. Turbo codes are linear codes. Linear codes are codes for which the modulo sum of two valid code words is also a valid codeword. A "good" linear code is one that has mostly high-weight code words. The weight or Hamming weight of a codeword is the number of ones that it contains, e.g., the Hamming weight of codeword '000' and '101' is 0 and 2 respectively. High-weight code words are desirable because it means that they are more distinct, and thus the decoder will have an easier time distinguishing among them. While a few low-weight code words can be tolerated, the relative frequency of their occurrence should be minimized.

The choice of the interleaver is crucial in the code design. Interleaver is used to scramble bits before being input to the second encoder. This makes the output of one encoder different from the other encoder. Thus, even if one of the encoders occasionally produces a low-weight, the probability of both the encoders producing a low-weight output is extremely small. This improvement is known as interleaver gain. Another purpose of interleaving is to make the outputs of the two encoders uncorrelated from each other. Thus, the exchange of information between the two decoders while decoding yields more reliability. There are different types of interleavers, e.g., row column interleaver, helical interleaver, odd-even interleaver, etc.

To summarize, it can be said that turbo codes make use of three simple ideas:

- Parallel concatenation of codes to allow simpler decoding
- Interleaving to provide better weight distribution
- Soft decoding to enhance decoder decisions and maximize the gain from decoder interaction.

3.1 Turbo code encoder

The fundamental turbo code encoder is built using two identical recursive systematic convolutional (RSC) encoders concatenated in parallel [9]. Convolutional codes are usually described using two parameters: the code rate *r* and the constraint length *K*. The code rate *k/n*, is expressed as a ratio of the number of bits into the convolutional encoder *k* to the number of channel symbols output by the convolutional encoder *n* in a given encoder cycle. The constraint length parameter *K* denotes the length of the

convolutional encoder, i.e., the maximum number of input bits that either output can depend on. Constraint length *K* is given as

$$K = m + 1 \tag{3.1}$$

Where *m* is the maximum number of stages (memory size) in any shift register. The shift registers store the state information of the convolutional encoder and the constraint length relates the number of bits upon which the output depends.

In turbo code encoder, both the RSC encoders are of short constraint length in order to avoid excessive decoding complexity. An RSC encoder is typically of rate *r = 1/2* and is termed a component encoder. The two component encoders are separated by an interleaver. The output of the turbo encoder consists of the systematic input data and the parity outputs from two constituent RSC encoders. The systematic outputs from the two RSC encoders are not needed because they are identical to each other (although ordered differently) and to the turbo code input. Thus the overall code rate becomes *r = 1/3*. Figure 3.1 shows the fundamental turbo code encoder [9].

The first RSC encoder outputs the systematic output c_1 and recursive convolutional encoded output sequence c_2 whereas the second RSC encoder discards its systematic sequence and only outputs the recursive convolutional encoded sequence c_3.

3.1.1 Recursive Systematic Convolutional (RSC) encoder

The RSC encoder is obtained from the conventional non-recursive non-systematic convolutional encoder by feeding back one of its encoded outputs to its input. Figure 3.2 shows a conventional rate *r = 1/3* convolutional encoder with constraint length *K=3*.

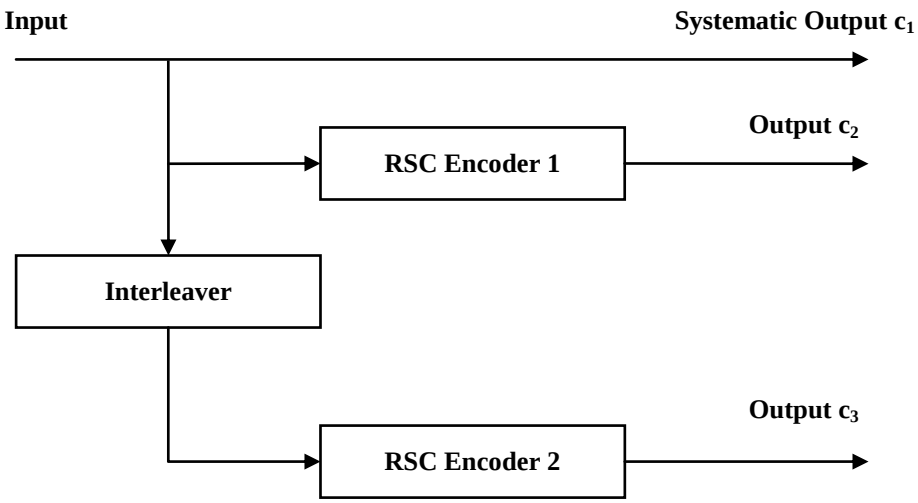


Figure 3.1 Fundamental turbo code encoder.

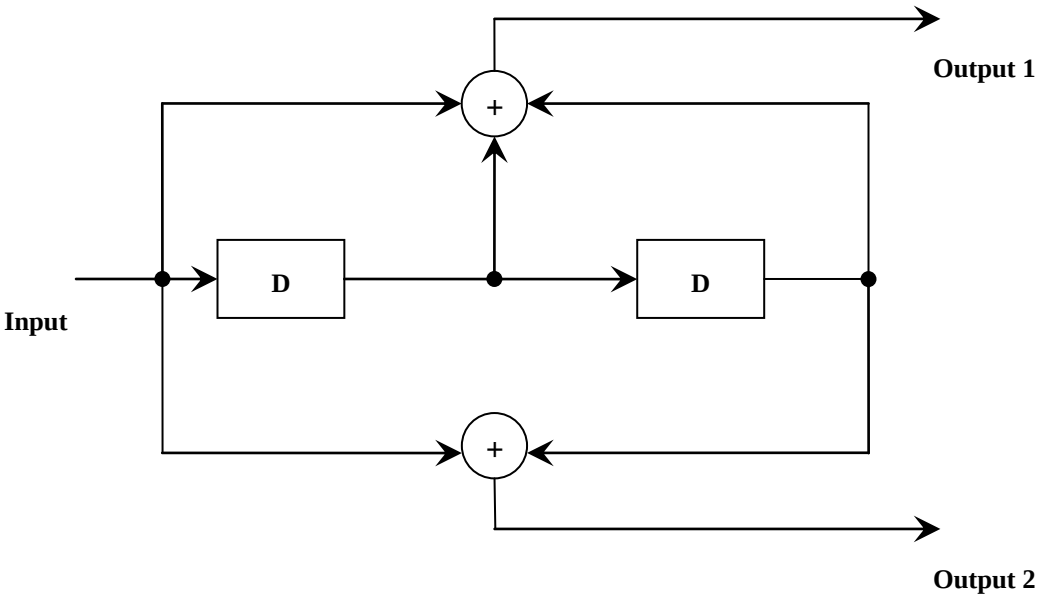


Figure 3.2 Conventional convolutional encoder with r = 1/2 and K = 3.

For each adder in the convolutional encoder, a generator polynomial is defined. It shows the hardware connections of the shift register taps to the modulo-2 adders. A "1" represents a connection and a "0" represents no connection. The generator polynomials for the above conventional convolution encoder are given as g_1= [111] and g_2 = [101] where the subscripts 1 and 2 denote the corresponding output terminals. The generator matrix of the convolutional encoder is a *k*-by-*n* matrix. The element in the i^{th} row and j^{th} column indicates how the i^{th} input contributes to the j^{th} output. The generator matrix of the above convolutional encoder is given in equation (3.2).

$$G = [\, g_1 \,,\, g_2 \,] = [111 \,, 101] \tag{3.2}$$

The conventional encoder can be transformed into an RSC encoder by feeding back the first output to the input. The generator matrix of the encoder then becomes

$$G = [\, 1 \,, \frac{g_2}{g_1} \,] \tag{3.3}$$

Where 1 denotes the systematic output, g_2 denotes the feed forward output, and g_1 is the feedback to the input of the RSC encoder. Figure 3.3 shows the resulting RSC encoder.

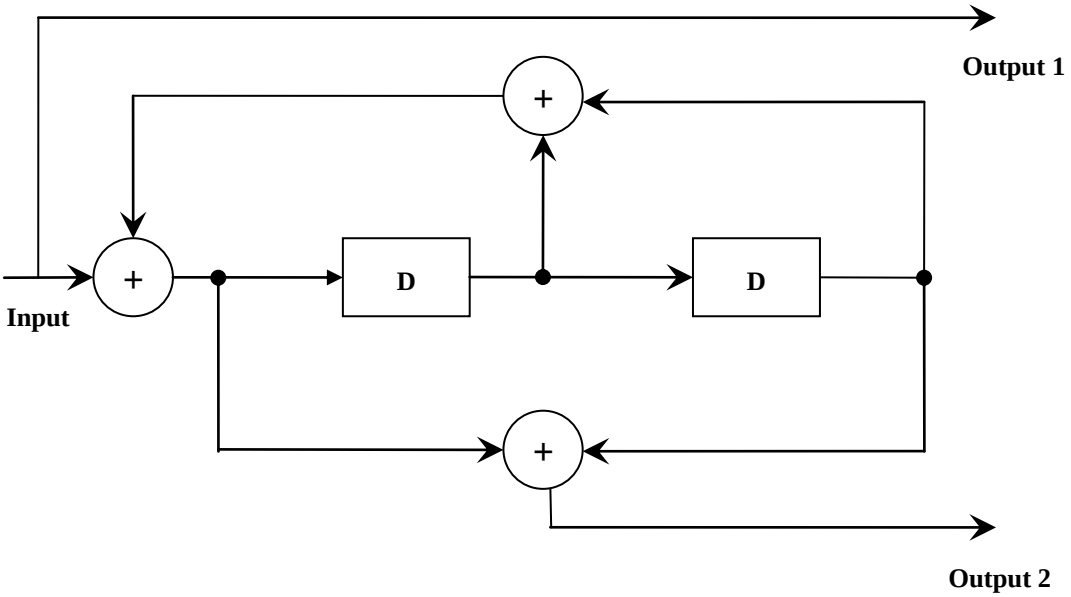


Figure 3.3 RSC encoder obtained from the conventional convolution encoder with r = 1/2 and K = 3.

It was suggested in [11] that good codes can be obtained by setting the feedback of the RSC encoder to a primitive polynomial, because the primitive polynomial generates maximum-length sequences which adds randomness to the turbo code.

3.1.2 Representation of turbo codes

Turbo codes are often viewed as finite state machines and are represented using state diagrams and trellis diagrams. The contents in the memory elements of a coder represent its state. For the rate $r = 1/2$ convolutional encoder of Figure 3.2 with constraint length $K=3$, the number of memory elements is $L=K-1 =2$. The input bit can be either one or zero so the size M of the input alphabet 2. The number of possible states of the coder is then $M^L = 2^2 = 4$ [12].

The operation of a convolutional coder can be represented by a state diagram which consists of a set of nodes S_j representing the possible states of the encoder where $j \in \{ 0 \ldots M^L \}$. The nodes are connected by branches and are labelled by the input symbol and the corresponding output symbol. Figure 3.4 shows the state diagram of the rate $r = 1/2$ convolutional encoder of Figure 3.2. The state diagram has four states $S = \{ 00, 01, 10, 11\}$. The arrows represent the transition from one state to other and the labelling on arrow gives the input bit (1 or 0) and the output of the encoder.

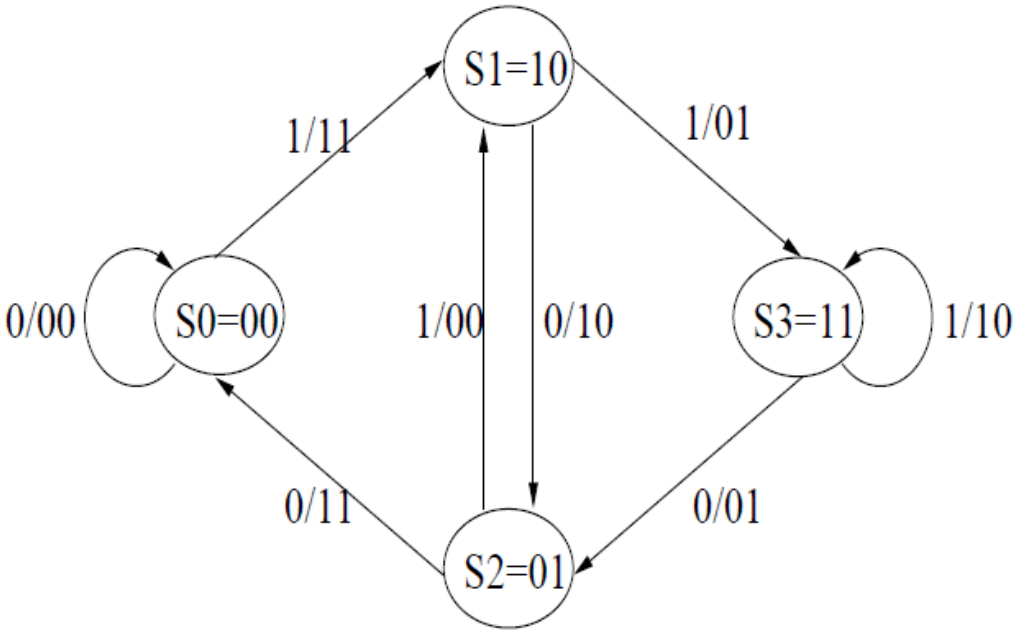


Figure 3.4 State diagram of rate r = 1/2 constraint length K = 3 convolutional encoder

The state diagram can be expanded into the trellis diagram. All state transitions at each time step are explicitly shown in the diagram to retain the time dimension. The trellis diagram is very convenient for describing the behaviour of the corresponding decoder. Figure 3.5 shows the trellis diagram for the encoder in Figure 3.2 [13].

The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one whereas the dotted lines represent state transitions when the input bit is a zero. The labels on the branch represent the output bits.

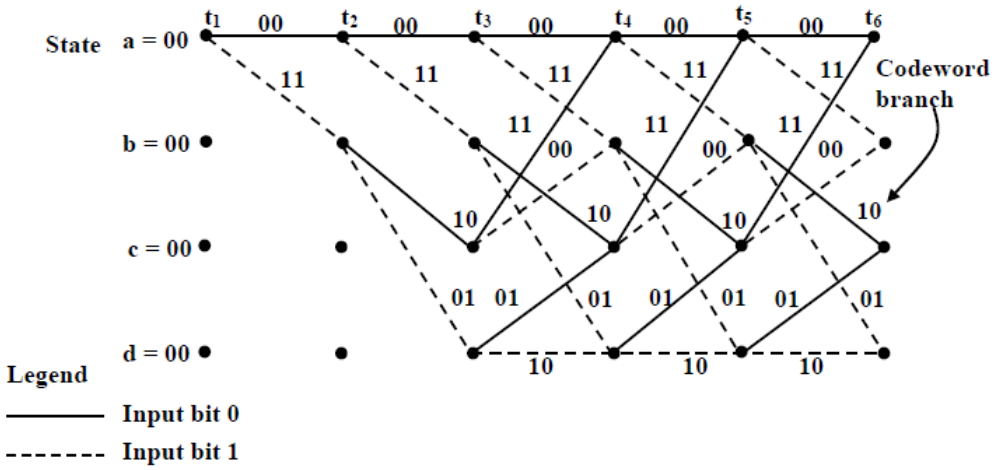


Figure 3.5 Trellis diagram for the encoder in Figure 3.2 [13].

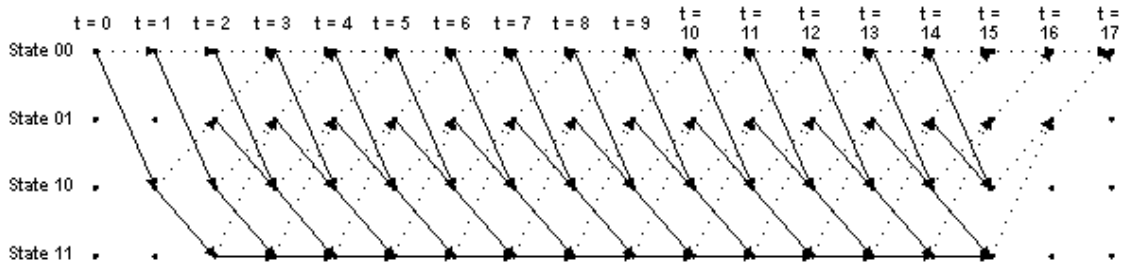


Figure 3.6 Trellis termination for the encoder in Figure 3.2 [14].

Consider the trellis diagram for encoding 15 input bits as shown in Figure 3.6 [14]. The trellis is in state *'00'* at the beginning. i.e., *t=0*. The trellis shows the next two states at time *t=1* with input bit *1* and *0*. At the end of the 15 bit input, the encoder can be in any of the states. However, for better performance of the decoder, both the initial and final states of the encoder should be known. Thus, it is desirable to bring the encoders to a known state after the entire input has been encoded. For this purpose, trellis termination is performed by passing *m=k*-1 tail bits from the constituent encoders bringing them to all zeros state. This brings the trellis to the initial all zero state as well. This is known as trellis termination.

3.1.3 Trellis termination

Unlike conventional convolutional codes which always use a stream of zeros as tail bits, the tail bits of a RSC depend on the state of the encoder when all the data bits have been encoded [15]. Also because of the presence of interleaver between the two encoders, the final states of the two component encoders will be different. Thus, the trellis termination bits for the two encoders will also be different and an RSC cannot be brought to an all zero state simply by passing a stream of zeros through it. However, this can be done by using the feedback bit as the encoder input. This is done by using a switch at the input as shown in Figure 3.7.

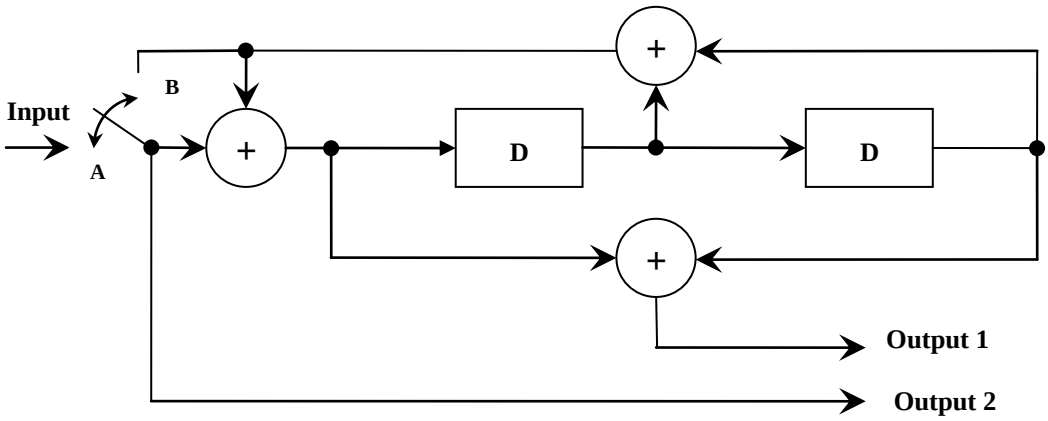


Figure 3.7 The trellis termination strategy for RSC encoder.

The switch is in position *A* while encoding the input sequence and is switched to position *B* at the end of the input bit sequence for termination of trellis. The XOR of the bit with itself will be zero (output of left most XOR) and thus the encoder will return to all zero state after *m=k-1* clock cycles by inputting the *m=k-1* feedback bits generated immediately after all the data bits have been encoded. Thus, a feedback is used for terminating the trellis bringing the encoder to the all zero state.

3.1.4 Recursive convolutional encoders vs. Non-recursive encoder

The recursive convolutional encoders are better suited for turbo codes as compared to non-recursive encoders because they tend to produce higher weight code words. Consider a rate $r = 1/2$ constraint length $K = 2$ non-recursive convolutional encoder with generator polynomial g_1 = [11] and g_2 = [10] as shown in Figure 3.8. The corresponding RSC encoder with generator matrix G = [1, g_2 / g_1] is shown in Figure 3.9.

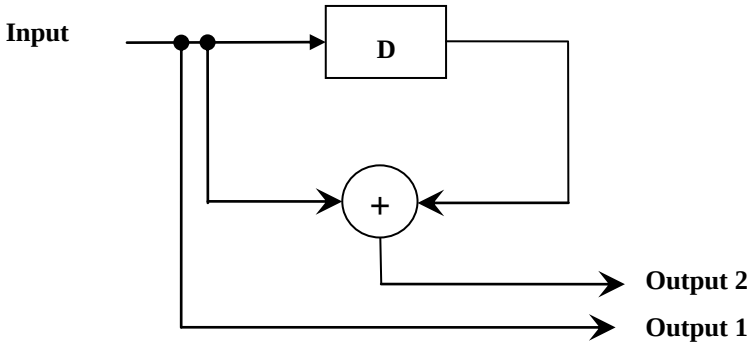


Figure 3.8 Non-recursive r=1/2 and K=2 convolutional encoder with input and output sequences.

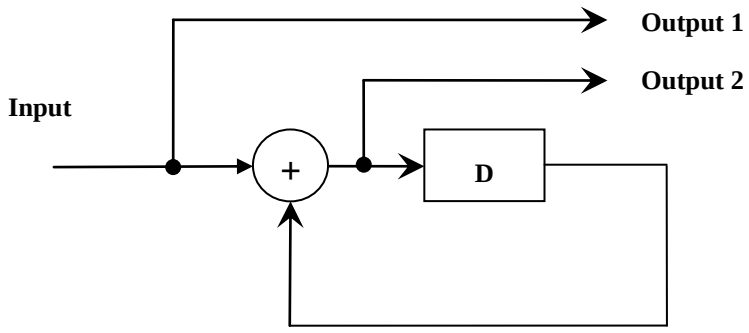


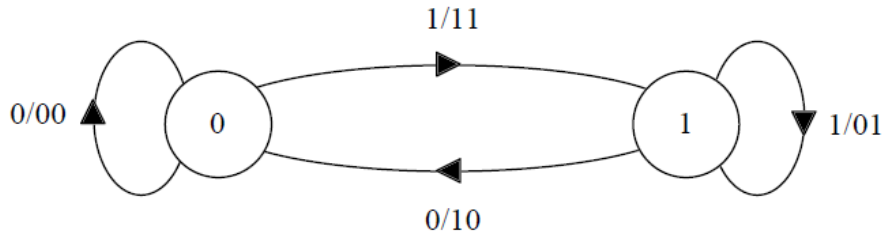
Figure 3.9 Recursive r = 1/2 and K = 2 convolutional encoder of Figure 3.8 with input and output sequences.

Table 3.1 Input and output sequences for convolutional encoders of Figure 3.8 and Figure 3.9.

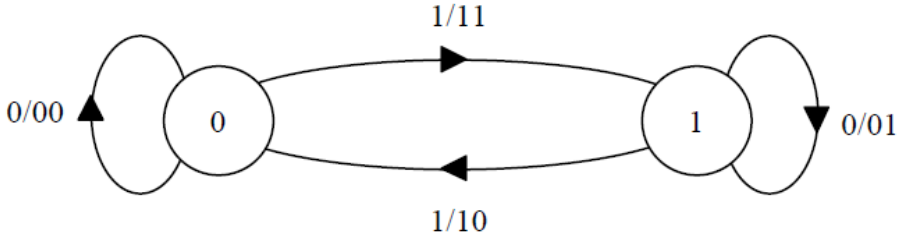
Encoder Type	Input	Output 1 c_1	Output 2 c_2	Weight
Non- RSC	1000	1000	1100	3
RSC	1000	1000	1111	5

Table 3.1 shows the output sequences corresponding to the same input sequence given to the two encoders. The non-recursive convolutional encoder produces output c_1 = [1100] and c_2 = [1000], thus it has a weight of *3*. On the other hand, the recursive convolution encoder outputs c_1= [1000] and c_2= [1111] which has a weight of *5*. Thus, a recursive convolutional encoder tends to produce higher weight code words as compared to non-recursive encoder, resulting in better error performance. For turbo codes, the main purpose of implementing RSC encoders as component encoders is to utilize the recursive nature of the encoders and not the fact that the encoders are systematic [16].

Figure 3.10 shows the state diagram of the non-recursive and recursive encoders. Clearly, the state diagrams of the encoders are very similar. Also, the two encoders have the same minimum free distance and can be described by the same trellis structure [9]. However, these two codes have different BERs as the BER depends on the input-output correspondence of the encoders [17]. It has been shown that the BER for a recursive convolutional code is lower than that of the corresponding non-recursive convolutional code at low signal-to-noise ratios E_b/N_o [9][17].



(a) State diagram of the non-recursive encoder in Figure 3.8.



(b) State diagram of recursive encoder in Figure 3.9.

Figure 3.10 The state diagram of recursive and non recursive encoders.

3.1.5 Concatenation of codes

In coding theory, concatenated codes form a class of error correcting codes obtained by combining an inner code with an outer code. The main purpose of concatenated codes is to have larger block length codes with exponentially decreasing error probability [18]. There are two types of concatenation, namely serial and parallel concatenations. In serial concatenation, the blocks are connected serially such that the output of one block is input to the next block and so on.

An interleaver is often used between the encoders in both serial and parallel concatenated codes to improve burst error correction capacity and increase the randomness of the code. In case of serial concatenation, the output of encoder 1 is interleaved before being input to encoder 2 whereas in parallel concatenation, the input data is interleaved before being input to the second encoder. The serial concatenated coding scheme is shown in Figure 3.11.

The total code rate for serial concatenation is the product of the code rates of the constituent encoders and is given as [18].

$$r_{total} = r_1 \times r_2 = \frac{k_1}{n_1} \times \frac{k_2}{n_2} = \frac{k_1 k_2}{n_1 n_2} \tag{3.4}$$

For parallel concatenation, the blocks are connected in parallel. The parallel concatenation scheme is shown in Figure 3.12. The total code rate for parallel concatenation is calculated as

$$\frac{1}{r_{total}} = \frac{1}{r_1} + \frac{1}{r_2} = \frac{1}{k/n_1} + \frac{1}{k/n_2} = \frac{n_1}{k} + \frac{n_2}{k} = \frac{n_1 + n_2}{k}$$

$$\Rightarrow r_{total} = \frac{k}{n_1 + n_2} \tag{3.5}$$

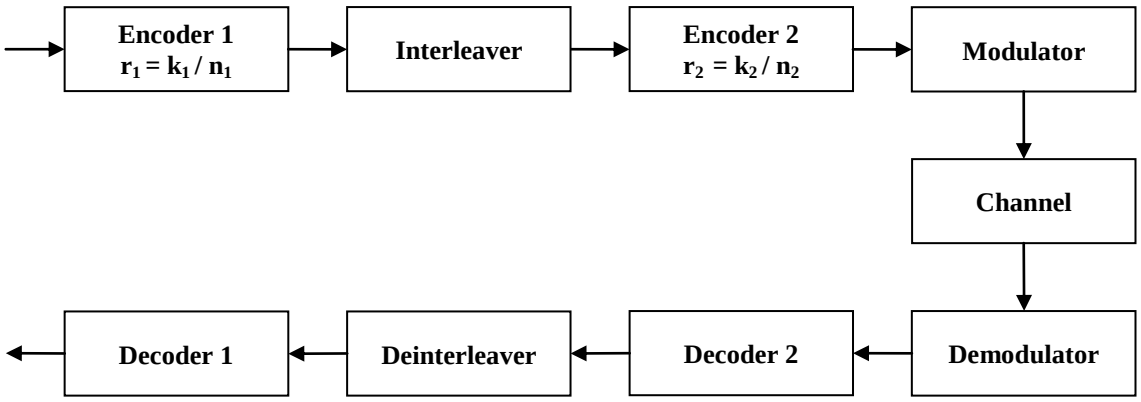


Figure 3.11 Serial concatenated code.

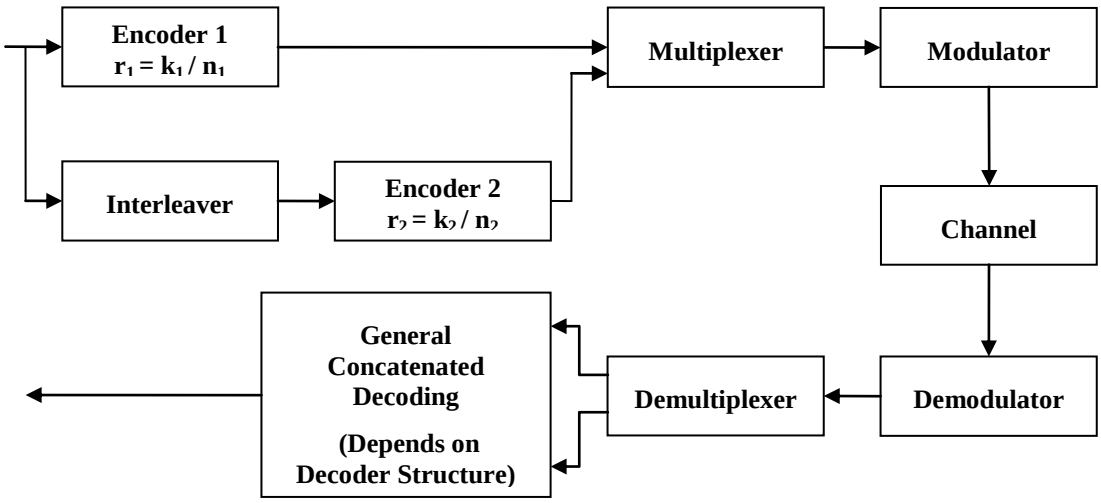


Figure 3.12 Parallel concatenated code.

Turbo codes use the parallel concatenated encoding scheme in which the two RSC encoders are connected in parallel separated by an interleaver. However, the turbo code decoder is based on the serial concatenated decoding scheme. The performance of serial concatenated decoders is better than the parallel concatenated decoding scheme as the serial concatenation scheme has the ability to share information between the concatenated decoders. On the other hand, the decoders for the parallel concatenation scheme are primarily decoding independently.

3.1.6 Interleaver design

Turbo codes use an interleaver between two component encoders. The purpose of using the interleaver is to provide randomness to the input sequences and increase the weight of the code words.

Consider a constraint length $K = 2$ rate $r = 1/2$ convolutional encoder as shown earlier in Figure 3.9. The input sequence x_i produces output sequences c_{1i} and c_{2i} respectively. The input sequences x_1 and x_2 are different permuted sequences of x_0. Table 3.2 shows the resulting code words and weights.

Table 3.2 Input and Output Sequences for Encoder in Figure 3.9.

	Input Sequence	Output Sequence c_{1i}	Output Sequence c_{2i}	Codeword Weight
x_0	1100	1100	1000	3
x_1	1010	1010	1100	4
x_2	1001	1001	1110	5

The values in Table 3.2 show that the codeword weight can be increased by permuting the input sequence using an interleaver. The interleaver affects the performance of turbo codes by directly affecting the distance properties of the code [19]. The BER of a turbo code is improved significantly by avoiding low-weight code words. The choice of the interleaver is important in the turbo code design. The optimal interleaver is the one that produces fewest low weight coded sequences. The algorithm shows the basic interleaver design concept:

- Generate a random interleaver.
- Generate all possible input information sequences.
- Determine the resulting code words for all possible input information sequences. Determine the weight of the code words to find the weight distribution of the code.
- From the collected data, determine the minimum codeword weight and the number of code words with that weight.

Repeat the algorithm for a reasonable number of times. By comparison of the data, the interleaver with the largest minimum codeword weight and lowest number of code words with that weight is selected.

A number of interleavers are used in turbo codes [20]. A few of them are discussed below in detail.

3.1.6.1 Matrix interleaver

The matrix interleaver is the most commonly used interleaver in communication systems. It writes data in a matrix columnwise from top to bottom and left to right without repeating or omitting any of the data bits. The data is then read out rowwise from left to right and top to bottom. For example, if the interleaver uses a 2-by-3 matrix to do its internal computations, then for an input of [A B C D E F], the interleaver matrix is

$$\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$$

The interleaved output is [A D B E C F]. At the deinterleaver the data is written in columnwise fashion and read rowwise to obtain the original data sequence.

3.1.6.2 Random (Pseudo-Random) interleaver

The random interleaver uses a fixed random permutation and maps the input sequence according to the permutation order. Consider an input sequence of length $L=8$ and the random permutation pattern be [2 3 5 6 7 4 8 1]. If the input sequence is [A B C D E F G H], the interleaved sequence will be [B C E F G D H A]. At the deinterleaver, the reverse permutation pattern is used to deinterleave the data.

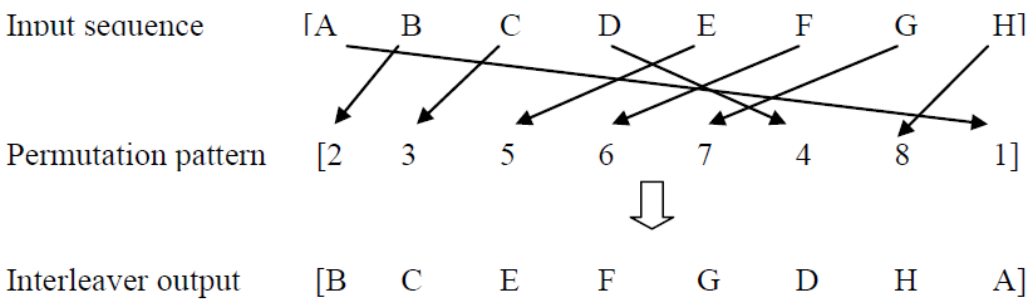


Figure 3.13 The interleaving using random interleaver.

3.1.6.3 Circular-shifting interleaver

The permutation p of the circular-shifting interleaver is defined by

$$p(i) = (bi + s) mod\ L \tag{3.7}$$

Where *i* is the index, *b* is the step size, *s* is the offset and *L* is the size of the interleaver. The step size *b* should be less than *L* and *b* should be relatively prime to *L* [21]. Figure 3.14 illustrates a circular-shifting interleaver with *L=8*, *b=3*, and *s=0*.

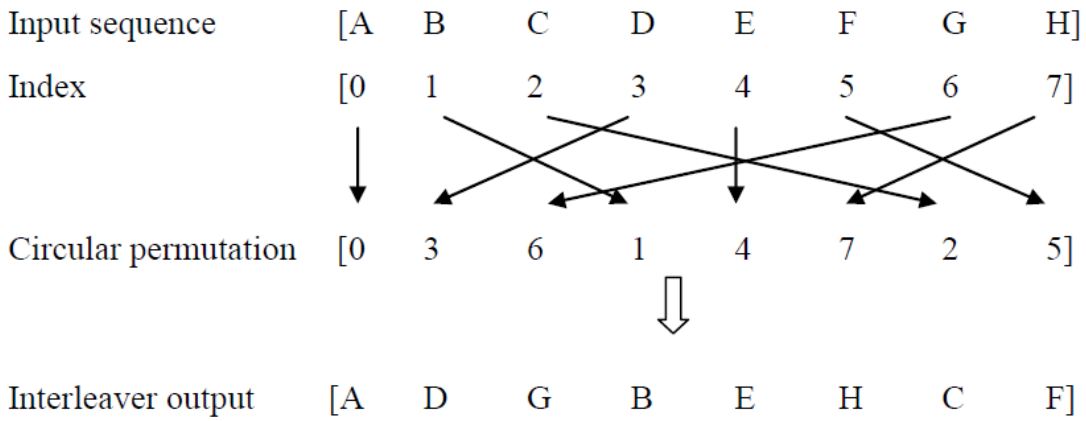


Figure 3.14 The interleaving using circular shifting interleaver.

It can be seen that the adjacent bit separation is either 3 or 5. This type of interleaver is suited for permuting weight-2 input sequences with low codeword weights into weight-2 input sequences with high codeword weights. However, because of the regularity of adjacent bit separation (3 or 5 in the above case), it is difficult to permute input sequences with weight higher than 2 using this interleaver [21].

3.1.6.4 Odd-Even interleaver design

The code rate of the turbo code can be changed by puncturing the output sequence. Puncturing is the process of removing some of the parity bits after encoding. This has the same effect as encoding with an error-correction code with a higher rate, or less

redundancy. However, with puncturing the same decoder can be used regardless of how many bits have been punctured, thus puncturing considerably increases the flexibility of the system without significantly increasing its complexity. A pre-defined pattern of puncturing is used at the encoder end. The inverse operation, known as depuncturing, is implemented by the decoder. By puncturing the two coded output sequences of a rate *r = 1/3* turbo code, a rate *r = 1/2* turbo code can be obtained. However, by simple puncturing the two coded output sequences there is a possibility that both the coded bits corresponding to a systematic information bits be punctured. Due to this, the information bit will have none of its coded bits in the sequence and vice versa. As a result, the performance of the turbo decoder degrades in case of error for an unprotected information bit. This can be avoided by using an odd-even interleaver design. First, the bits are left uninterleaved and encoded, but only the odd-positioned coded bits are stored. Then, the bits are scrambled and encoded, but now only the even-positioned coded bits are stored. As a result, each information bit will now have exactly one of its coded bits [22]. This guarantees an even protection of each bit of information, a uniform distribution of the error correction capability and better performance of the code.

3.2 Turbo code decoder

In traditional decoding approach, demodulation is based on hard decision of the received symbol. This discrete value is then passed on to the error control decoder. This approach has some disadvantages. The decoder cannot make use of the certainty of information available to it while decoding [9].

A similar but better rule is to take into account the *a priori* probabilities of the input. If the +1 symbol has a probability of 0.9 and if the symbol falls in negative decision range, the Maximum Likelihood Decoder (MLD) will decide it as -1. It does not take into account the 0.9 probability of symbol being +1. A detection method that does take this conditional probability into account is the Maximum *a posteriori* probability (MAP) algorithm. It is also known as the minimum error rule [23].

Another algorithm used for turbo decoding is the soft output Viterbi algorithm (SOVA). It uses Viterbi algorithm but with soft outputs instead of hard.

3.2.1 Map decoder

The process of MAP decoding includes the formation of *a posteriori* probabilities (APP) of each information bit followed by choosing the data bit value corresponding to MAP probability for that data bit. While decoding, the decoder receives as input a “soft” (i.e. real) value of the signal. The decoder then outputs for each data bit an estimate expressing the probability that the transmitted data bit was equal to one indicating the reliability of the decision. In the case of turbo codes, there are two decoders for outputs from both encoders. Both decoders provide estimates of the same set of data bits, but in

a different order due to the presence of interleaver. Information exchange is iterated a number of times to enhance performance. During each iteration, the estimates are re-evaluated by the decoders using information from the other decoder. This allows the decoder to find the likelihood as to what information bit was transmitted at each time instant. In the final stage, hard decisions will be made, i.e., each bit is assigned the value 1 or 0 [24].

The APPs are used to calculate the likelihood ratio by taking their ratio. The logarithmic form of likelihood ratio is the log likelihood ratio (LLR).

$$\Lambda(\hat{d}_k) = \frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}} \tag{3.8}$$

$$L(\hat{d}_k) = log\left[\frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}}\right] \tag{3.9}$$

Where $\Lambda(\hat{d}_k)$ is the likelihood ratio, $L(\hat{d}_k)$ is the LLR. The term $\lambda_k^{i,m}$ is described as the joint probability that data $d_k = i$ and state $S_k = m$, observed from time k =1 to N and conditioned on the received corrupted binary sequence R_1^N which has been transmitted through the channel, demodulated and presented to the decoder in soft decision form.

$$\lambda_k^{i,m} = P(d_k = i, S_k = m | R_1^N) \tag{3.10}$$

The sequence R_1^N can be written as

$$R_1^N = \{R_1^{k-1}, R_k, R_{k+1}^N\} \tag{3.11}$$

Substituting equation (3.11) in equation (3.10), we get

$$\lambda_k^{i,m} = P\left(d_k = i, S_k = m | R_1^{k-1}, R_k, R_{k+1}^N\right) \tag{3.12}$$

The Bayes's theorem gives the conditional probabilities as

$$\begin{aligned} P(A|B,C,D) &= \frac{P(A,B,C,D)}{P(B,C,D)} = \frac{P(B|A,C,D)P(A,C,D)}{P(B,C,D)} \\ &= \frac{P(B|A,C,D)P(D|A,C)P(A,C)}{P(B,C,D)} \end{aligned} \tag{3.13}$$

Applying the results of equation (3.13) to equation (3.12) yield,

$$\lambda_k^{i,m} = \frac{P\left(R_1^{k-1} \middle| R_k, R_{k+1}^N, d_k = i, S_k = m\right) P\left(R_{k+1}^N \middle| d_k = i, S_k = m, R_k\right) P(d_k = i, S_k = m, R_k)}{P(R_1^{k-1}, R_k, R_{k+1}^N)}$$

$$= \frac{P\left(R_1^{k-1}\middle|R_k^N, d_k = i, S_k = m\right)P\left(R_{k+1}^N\middle|d_k = i, S_k = m, R_k\right)P(d_k = i, S_k = m, R_k)}{P(R_1^N)} \quad (3.14)$$

Equation (3.14) will now be explained in terms of the forward state metric, reverse state metric and the branch metric.

3.2.1.1 The state metrics and the branch metric

- FORWARD STATE METRIC

Consider the first term on the right side of equation (3.14)

$$P\left(R_1^{k-1}\middle|R_k^N, d_k = i, S_k = m\right) \quad (3.15)$$

For i =0,1 and time k and state $S_k = m$ implies that the events before time k are not influenced by events after time k, i.e., the future does not affect the past. Thus the two terms R_k^N and $d_k = i$ are irrelevant and $P\left(R_1^{k-1}\right)$ is independent of these terms. However, since the encoder has memory, the state $S_k = m$ is based on the past so this term is relevant. This simplifies equation 3.15 as

$$P\left(R_1^{k-1}\middle|S_k = m\right) \triangleq \alpha_k^m \quad (3.16)$$

The equation represents the forward state metric at time k as being a probability of the past sequence that is dependent only on the current state $S_k = m$ induced by the sequence.

- REVERSE STATE METRIC

The second term of equation (3.14) represents the reverse state metric β_k^m at time k and state $S_k = m$.

$$P(R_{k+1}^N|d_k = i, S_k = m, R_k) = P\left(R_{k+1}^N\middle|S_{k+1} = f(i,m)\right) \triangleq \beta_{k+1}^{f(i,m)} \quad (3.17)$$

Where *f (i,m)* is the next state given an input i and state m and $\beta_{k+1}^{f(i,m)}$ is the reverse state metric at time k+1 and state *f(i,m).* Equation (3.17) represents the reverse state metric β_{k+1}^m at future time k+1 as being a probability of the future sequence which depends on the state at future time k+1. The future state *f (i,m)* in turn is a function of the input bit and the state at current time k.

- BRANCH METRIC

The third term in equation (3.14) is defined as the branch metric $\delta_k^{i,m}$at time *k* and state *m*.

$$P(d_k = i, S_k = m, R_k) \triangleq \delta_k^{i,m} \tag{3.18}$$

Substituting equation (3.16), (3.17), (3.18) in equation (3.14) yields

$$\lambda_k^{i,m} = \frac{\alpha_k^m \, \beta_{k+1}^{f(i,m)} \delta_k^{i,m}}{P(R_1^N)} \tag{3.19}$$

Equation (3.8) and (3.9) can be expressed in terms of equation (3.19) as

$$\Lambda(\hat{d}_k) = \frac{\sum_m \alpha_k^m \, \beta_{k+1}^{f(1,m)} \delta_k^{1,m}}{\sum_m \alpha_k^m \, \beta_{k+1}^{f(0,m)} \delta_k^{0,m}} \tag{3.20}$$

$$L(\hat{d}_k) = log\left[\frac{\sum_m \alpha_k^m \, \beta_{k+1}^{f(1,m)} \delta_k^{1,m}}{\sum_m \alpha_k^m \, \beta_{k+1}^{f(0,m)} \delta_k^{0,m}}\right] \tag{3.21}$$

Equations (3.20) and (3.21) represent the likelihood ratio $\Lambda(\hat{d}_k)$ and the LLR $L(\hat{d}_k)$ of the k^{th} data bit respectively.

3.2.1.2 Calculating the forward state metric

The forward state metric in equation (3.16) can be expressed as the summation of all possible transition probabilities from time *k*-1, as follows [24]

$$P(R_1^{k-1} | S_k = m) \triangleq \alpha_k^m = \sum_{m'} \sum_{j=0}^{1} P(d_{k-1} = j, S_{k-1} = m', R_1^{k-1} | S_k = m)$$

$$\alpha_k^m = \sum_{m'} \sum_{j=0}^{1} P(d_{k-1} = j, S_{k-1} = m', R_1^{k-2}, R_{k-1} | S_k = m) \tag{3.22}$$

Applying Bayes's theorem yields

$$\begin{aligned} \alpha_k^m = \sum_{m'} \sum_{j=0}^{1} & P(R_1^{k-2} | S_k = m, d_{k-1} = j, S_{k-1} = m', R_{k-1}) \\ & \times \; P(\, d_{k-1} = j, S_{k-1} = m', R_{k-1} | S_k = m) \end{aligned} \tag{3.23}$$

As the knowledge about state *m´* and the input *j* at time *k*-1 completely defines the path resulting in state $S_k = m$, so the equation can be simplified to:

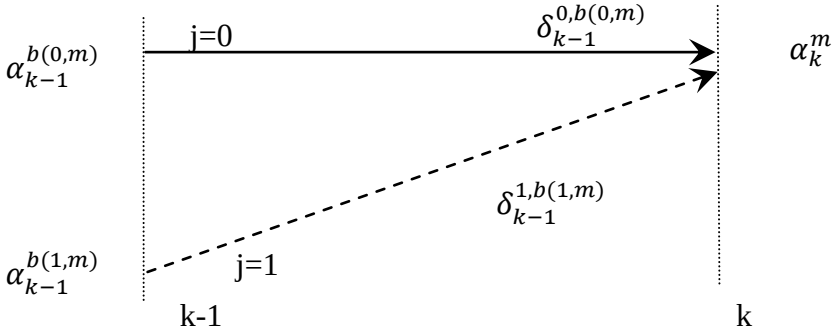


Figure 3.15 The graphical representation for calculating the forward state metric.

$$\alpha_k^m = \sum_{j=0}^{1} P\left(R_1^{k-2} \middle| S_k = b(j,m)\right) P(d_{k-1} = j, S_{k-1} = b(j,m), R_{k-1}) \tag{3.24}$$

Where *b(j, m)* is the state going backward in time from state *m*, via the previous branch corresponding to input *j*. Equation (3.24) can be simplified using equations (3.16) and (3.18) .

$$\alpha_k^m = \sum_{j=0}^{1} \alpha_{k-1}^{b(j,m)} \ \delta_{k-1}^{j,b(j,m)} \tag{3.25}$$

Figure 3.15 represents equation (3.25) in terms of transitions from state *k-1* to state *k.*

3.2.1.3 Calculating the reverse state metric

The reverse state metric was given in equation (3.17) as

$$\beta_{k+1}^{f(i,m)} = P\left(R_{k+1}^N \middle| S_{k+1} = f(i,m)\right)$$

Using this equation, the reverse state metric for state *m* at time *k* is given as:

$$\beta_k^m = P(R_k^N | S_k = m) \quad = \quad P(R_k \, R_{k+1}^N | S_k = m) \tag{3.26}$$

Equation (3.26) can be expressed in terms of summation of all possible transition probabilities to time *k+1* as follows [24]

$$\beta_k^m = \sum_{m'} \sum_{j=0}^{1} P(d_k = j, S_{k+1} = m', R_k, R_{k+1}^N | S_k = m) \tag{3.27}$$

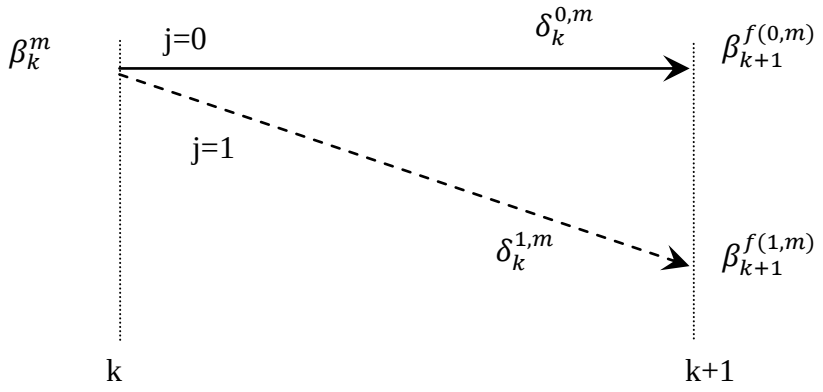


Figure 3.16 The graphical representation for calculating reverse state metric.

Using Bayes's theorem

$$\beta_k^m = \sum_{m'} \sum_{j=0}^{1} P(R_{k+1}^N | S_k = m, d_k = j, S_{k+1} = m', R_k) \times P(d_k = j, S_{k+1} = m', R_k | S_k = m) \tag{3.28}$$

In the first term of equation (3.28), the terms $S_k = m$ and $d_k = j$ define the path given an input j and state m resulting in state $S_{k+1} = f(j,m)$. Thus, the term $S_{k+1} = m'$ can be replaced with $S_k = m$ in the second term of equation (3.28).

$$\beta_k^m = \sum_{j=0}^{1} P\left(R_{k+1}^N | S_{k+1} = f(j,m)\right) \times P(d_k = j, S_k = m, R_k) \tag{3.29}$$

Using equation (3.17) and (3.18), equation (3.29) becomes

$$\beta_k^m = \sum_{j=0}^{1} \delta_k^{j,m} \beta_{k+1}^{f(j,m)} \tag{3.30}$$

The equation represents the reverse state metric at time *k* as the weighted sum of state metrics from time *k+1* where the weighting consists of branch metrics associated with transitions corresponding to data bits 0 and 1. Figure 3.16 shows the graphical representation of calculating the reverse state metric.

3.2.1.4 Calculating the branch metric

The branch metric was given in equation (3.18) as

$$\delta_k^{i,m} = P(d_k = i, S_k = m, R_k)$$

The equation can be solved using Bayes's rule as

$$\delta_k^{i,m} = P(R_k|d_k = i, S_k = m)P(S_k = m|d_k = i)P(d_k = i) \tag{3.31}$$

Where R_k is the received noisy signal and it consists of both noisy data bits x_k and noisy parity bits y_k .The noise affects data and parity bits independently so the current state is independent of the current input and can therefore be any of the 2^v states where v is the number of memory elements in the convolutional code system. Thus the probability is given as

$$P(S_k = m|d_k = i) = \frac{1}{2^v} \tag{3.32}$$

Solving equation (3.31) using equation (3.32) yields

$$\delta_k^{i,m} = \frac{P(R_k|d_k = i, S_k = m)P(d_k = i)}{2^v} = \frac{P(x_k, y_k|d_k = i, S_k = m)P(d_k = i)}{2^v} \tag{3.33}$$

The term $P(d_k = i)$ is the *a priori* probability of d_k and is given as π_k^i , thus equation (3.33) becomes

$$\delta_k^{i,m} = \frac{P(x_k|d_k = i, S_k = m)P(y_k|d_k = i, S_k = m)\pi_k^i}{2^v} \tag{3.34}$$

Now, the probability density function (pdf) $p_{X_K}(x_k)$ of a random variable X_k is related to the probability of that random variable X_k taking on the value x_k as

$$P(X_k = x_k) = p_{X_K}(x_k)dx_k \tag{3.35}$$

For an AWGN channel with zero mean and variance σ^2, the probability terms in equation (3.34) are replaced with the pdf equivalents of equation (3.35) yielding

$$\delta_k^{i,m} = \frac{\pi_k^i}{2^v} exp\left[-\frac{1}{2}\left(\frac{x_k - u_k^i}{\sigma}\right)^2\right] dx_k \frac{1}{\sqrt{2\pi}\sigma} exp\left[-\frac{1}{2}\left(\frac{y_k - v_k^{i,m}}{\sigma}\right)^2\right] dy_k \tag{3.36}$$

Where u_k and v_k represent the transmitted data bits and parity bits respectively. The parameter u_k^i represents data that has no dependence on state m whereas the parameter $v_k^{i,m}$ represents parity which depends on the state m since the code has memory. Simplifying equation (3.36) gives [24]:

$$\delta_k^{i,m} = A_k \pi_k^i \, exp\left[\frac{1}{\sigma^2}\left(x_k u_k^i + y_k v_k^{i,m}\right)\right] \tag{3.37}$$

Substituting equation (3.37) in equation (3.8), we obtain:

$$\Lambda(\hat{d}_k) = \pi_k exp\left(\frac{2x_k}{\sigma^2}\right)\frac{\sum_m \alpha_k^m \beta_{k+1}^{f(1,m)} exp\left(\frac{y_k v_k^{1,m}}{\sigma^2}\right)}{\sum_m \alpha_k^m \beta_{k+1}^{f(0,m)} exp\left(\frac{y_k v_k^{0,m}}{\sigma^2}\right)} \tag{3.38a}$$

$$\Lambda(\hat{d}_k) = \pi_k exp\left(\frac{2x_k}{\sigma^2}\right)\pi_k^e \tag{3.38b}$$

Where π_k is the input *a prior* probability ratio π^1/π^0 at time k and π_k^e is the output extrinsic likelihood. The term π_k^e can be considered as the correction term that changes the input priori knowledge about a data bit due to coding and is passed from one decoder to the other during iterations. This improves the likelihood ratio for each data bit and minimizes the decoding errors. Taking log of equation (3.38b) yields the final LLR term.

$$L(\hat{d}_k) = L(d_k) + L_c(x_k) + L_e(\hat{d}_k) \tag{3.39}$$

The final soft number $L(\hat{d}_k)$ is made up of three LLR terms; the *a priori* LLR $L(d_k)$, the channel measurement LLR $L_c(x_k)$, and the extrinsic LLR $L_e(\hat{d}_k)$.

Implementing the MAP algorithm in terms of the likelihood ratios is very complex because of the multiplication operations that are required. However, by implementing it in the logarithmic domain, the complexity is greatly reduced.

4. THE UMTS TURBO CODE

Turbo codes are discussed in detail in chapter 3. This chapter will focus on the implementation of Turbo encoder and decoder for UMTS systems. The architecture of the encoder will be discussed in section 4.1. It explains the basic convolutional encoders and interleaver used in UMTS turbo code in detail. After coding, the systematic bits and code bits are arranged in a sequence, modulated using BPSK modulation and transmitted over the channel. The channel models and the decoder architecture are discussed in detail in section 4.2 and section 4.3 respectively.

4.1 UMTS turbo encoder

The UMTS turbo coder scheme is Parallel Concatenated Convolution Code (PCCC). It comprises of two constraint length $K = 4$ (8 state) RSC encoders concatenated in parallel. The overall code rate is approximately $r = 1/3$. Figure 4.1 shows a UMTS turbo encoder [26].

The two convolutional encoders used in the Turbo code are identical with generator polynomials [25].

$$g_0(D) = 1 + D^2 + D^3 \tag{4.1}$$

$$g_1(D) = 1 + D + D^3 \tag{4.2}$$

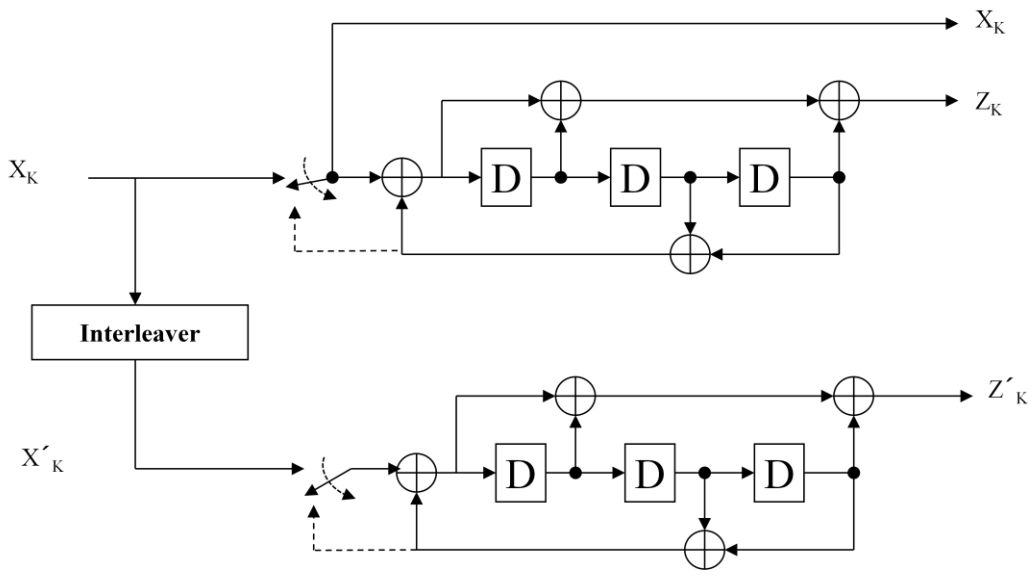


Figure 4.1 The UMTS turbo encoder [26].

Where g_0 and g_1 are the feedback and feed forward generator polynomials respectively.

The transfer function of each constituent convolutional encoder is:

$$G(D) = \left[1, \frac{g_1(D)}{g_0(D)}\right] \tag{4.3}$$

The data bits are transmitted together with the parity bits generated by two constituent convolutional encoders. Prior to encoding, both the convolutional encoders are set to all zero state, i.e., each shift register is filled with zeros. The turbo encoder consists of an internal interleaver which interleaves the input data bits $X_1, X_2 \ldots\ldots X_K$ *to* $X'_1, X'_2 \ldots\ldots X'_K$ which are then input to the second constituent encoder. Thus, the data is encoded by the first encoder in the natural order and by the second encoder after being interleaved. The systematic output of the second encoder is not used and thus the output of the turbo coder is serialized combination of the systematic bits X_k, parity bits from the first (upper) encoder Z_k and parity bits from the second encoder Z'_k for k = 1,2,…K.

$$X_1, Z_1, Z'_1, X_2, Z_2, Z'_2 \ldots\ldots\ldots\ldots\ldots X_K, Z_K, Z'_K \tag{4.4}$$

The size of the input data word may range from as few as 40 to as many as 5114 bits. If the interleaver size is equal to the input data size K the data is scrambled according to the interleaving algorithm, otherwise dummy bits are added before scrambling. After all the data bits K have been encoded, trellis termination is performed by passing tail bits from the constituent encoders bringing them to all zeros state. To achieve this, the switches in Figure 4.1 are moved in the down position. The input in this case is shown by dashed lines (input=feedback bit). Because of the interleaver, the states of both the constituent encoders will usually be different, so the tail bits will also be different and need to be dealt separately.

As constraint length $K=4$ constituent convolutional encoders are used, so the transmitted bit stream includes not only the tail bits $\{X_{k+1}, X_{k+2}, X_{k+3}\}$ corresponding to the upper encoder but also tail bits corresponding to the lower encoder $\{X'_{k+1}, X'_{k+2}, X'_{k+3}\}$. In addition to these six tail bits, six corresponding parity bits $\{Z_{k+1}, Z_{k+2}, Z_{k+3}\}$ and $\{Z'_{k+1}, Z'_{k+2}, Z'_{k+3}\}$ for the upper and lower encoder respectively are also transmitted. First, the switch in the upper (first) encoder is brought to lower (flushing) position and then the switch in the lower (second) encoder. The tail bits are then transmitted at the end of the encoded data frame. The tail bits sequence is:

$$X_{K+1}, Z_{K+1}, X_{K+2}, Z_{K+2}, X_{K+3}, Z_{K+3}, X'_{K+1}, Z'_{K+1}, X'_{K+2}, Z'_{K+2}, X'_{K+3}, Z'_{K+3} \tag{4.5}$$

The total length of the encoded bit sequence now becomes $3K+12$, $3K$ being the coded data bits and 12 being the tail bits. The code rate of the encoder is thus

r = K / (3K+12). However, for large size of input *K*, the fractional loss in code rate due to tail bits in negligible and thus, the code rate is approximated at 1/3.

4.1.1 Interleaver

Turbo code uses matrix interleaver [25]; [26]. Bits are input to a rectangular matrix in a rowwise fashion, inter-row and intra-row permutations are performed and the bits are then read out columnwise. The interleaver matrix can have 5, 10 or 20 rows and columns between 8 and 256 (inclusive) depending upon the input data size. Zeros are padded if the number of data bits is less than the interleaver size and are later pruned after interleaving.

The input to the interleaver is the input data bit sequence X_1, X_2 ….. ….X_K where *K* is the number of data bits. Interleaving is a complicated process and it comprises of a number of steps which are described below.

4.1.1.1 Bits input to the rectangular matrix

1. Determine number of rows *R* :

The number of rows can be either 5, 10 or 20 depending on the size of input data. If the input data size lies between 40 and 159, the number of rows *R* is five. If the input data size *K* is either between 160 and 200 or 481 and 530, the number of rows of the interleaver matrix *R* is ten. The interleaver matrix has twenty rows for size of input data *K* lying anywhere outside the range specified.

$$R = \begin{cases} 5 & if \;\; 40 \le K \le 159 \\ 10 & if \;\; 160 \le K \le 200 \;\; or \;\; 481 \le k \le 530 \\ 20 & if \; K = any \; other \; value \end{cases} \tag{4.6}$$

The rows are numbered from 0 to *R*-1 from top to bottom.

2. Determine number of columns *C* :

For determining *C*, first a prime root *p* is calculated which is later used in the intra-row permutations. If *K* lies in the range between 481 and 530 (inclusive), then both *C* and *p* are assigned a value 53. Otherwise, the value of *p* is selected from Table 4.1 such that the product of *R* and *(p+1)* is less than or equal to the number of input bits *K*. In this case, the number of columns C can have any of the three values *p-1*, *p* and *p+1* depending on the relation between number of data bits *K*, number of rows *R* and prime root *p*. Like the rows, the number of columns is also numbered from 0 to *C-1*.

$$C = \begin{cases} p-1 & if\ K \le R \times (p-1) \\ p & if\ R \times (p-1) < K \le R \times p \\ p+1 & if\ R \times p < K \end{cases} \quad (4.7)$$

Table 4.1 The list of prime number p and associated primitive root v.

p	*v*	*p*	*v*	*p*	*v*	*p*	*v*	*p*	*v*
7	3	47	5	101	2	157	5	223	3
11	2	53	2	103	5	163	2	227	2
13	2	59	2	107	2	167	5	229	6
17	3	61	2	109	6	173	2	233	3
19	2	67	2	113	3	179	2	239	7
23	5	71	7	127	3	181	2	241	7
29	2	73	5	131	2	191	19	251	6
31	3	79	3	137	3	193	5	257	3
37	2	83	2	139	2	197	2		
41	6	89	3	149	2	199	3		
43	3	97	5	151	6	211	2		

3. Writing data in rectangular matrix:

The data is written in the interleaver matrix row wise. The size of the interleaver matrix is $R \times C$. If the size of input stream is smaller than the interleaver size, dummy bits are padded which are later pruned away after interleaving. The interleaver matrix is

$$\begin{bmatrix} Y_1 & Y_2 & Y_3 & \dots\dots & Y_C \\ Y_{C+1} & Y_{C+2} & Y_{C+3} & \dots\dots & Y_{2C} \\ \vdots & \vdots & \vdots & & \vdots \\ Y_{((R-1)C+1)} & Y_{((R-1)C+2)} & Y_{((R-1)C+3)} & \dots\dots & Y_{R\times C} \end{bmatrix}$$

$Y_k = X_k \quad for\ k \le K$

$Y_k = 0, 1 \quad for\ k = (K+1), (K+2) \dots\dots (R \times C)$

Where K is the length of Input bit sequence and k =0, 1, 2.....K.

4.1.1.2 Intra row and inter row permutations

The permutation process consists of seven steps which have been discussed in detail below:

1. For the calculated value of prime root p, corresponding primitive toot v is selected from Table 4.1.

2. The base sequence *s* for intra row permutation is constructed by using the primitive root *v*, the prime root *p* and the previous value of base sequence. The value of base sequence is initialized to one.

$$s(j) = \left(v \times s(j-1)\right) mod\ p \quad for\ j = 0, 1, \ldots (p-2)\ and\ s(0) = 1 \tag{4.8}$$

3. The prime number sequence *q* is found out using the value of prime root *p* which is later used for the calculations of variables for intra-row permutation. The size of the sequence is equal to the number of rows of the interleaver matrix. The value of *q* is calculated using the greatest common divisor (gcd). The prime integer q_i *(i = 1, 2, ... R-1)* in the sequence is the least prime integer such that the gcd of *q* and *(p-1)* equals one. The value of *q* should be greater than 6. Also, the next value of *q* should be greater than the previous value in the sequence. The value of *q* is initialized to 1.

$$\gcd(q, p-1) = 1 \tag{4.9}$$

$$q_0 = 1\,, \qquad q_i > 6\ \ and\ \ q_i > q_{(i-1)}\ \ \ for\ i = 1, 2 \ldots .. R-1$$

4. The inter-row permutation pattern is selected depending on the number of input bits *K* and number of rows *R* of the interleaver matrix. The inter-row permutation pattern simply changes the ordering of rows without changing the ordering of elements within a row. If the number of rows is either five or ten, the inter-row permutation pattern is simply the flipping of rows or in other words a reflection around the centre row, e.g., if the number of rows is five, then the rows {0, 1, 2, 3, 4} are permuted to rows {4, 3, 2, 1, 0} respectively. Same is the case when the number of rows is ten. The rows {0, 1, 2, 3 ….9} become rows {9, 8, 7, 6, 5, 4, 3, 2, 1, 0} respectively. When the number of rows is twenty, there are two different cases based on the number of input bits. The rows {0, 1, 2,…19} become rows {20, 10, 15, 5, 1, 3, 6, 8, 13, 19, 17, 14, 18, 16, 4, 2, 7, 12, 9, 11} respectively, when the number of input bits *K* satisfies either of the two conditions, i.e., $2281 \leq K \leq 2480$ or $3161 \leq K \leq 3210$. For any other value of *K*, the permutation pattern is {20, 10, 15, 5, 1, 3, 6, 8, 13, 19, 11, 9, 14, 18, 4, 2, 17, 7, 16, 12}.

5. The prime integer sequence q_i is permuted to get r_i *(for i = 0, 1, ... R-1*) which is used in calculating the intra-row permutation patterns.

$$r_{T(i)} = q_i \quad for\ i = 0, 1 \ldots \ldots R-1 \tag{4.10}$$

Where *T*(*i*) refers to the i^{th} value of the inter-row permutation pattern calculated in the previous step.

6. Next, the intra-row permutation is performed depending on the relation between number of columns *C* and the prime root *p*. The intra-row permutation changes the

ordering of elements within a row without changing the ordering of the rows. The i^{th} *(i = 0, 1, ….R-1)* intra-row permutation is performed as:

$$U_i(j) = s\big((j \times r_i) mod(p-1)\big) \quad for\ j = 0,1, \dots (p-2) \tag{4.11}$$

Where $U_i(j)$ is the original bit position of the j^{th} permuted bit of i^{th} row. For example, the values *i*=3, j=4 and $U_i(j) = 15$ means that the bit at row 3 and column 4 now was originally at position 15 in the same row.

If the number of columns *C* is one less than the prime root *p*, then equation (4.11) is used for permuting the sequence. If number of columns *C* is equal to the prime root *p*, then the last column of i^{th} row $U_i(p-1)$ is assigned a value zero.

If the number of columns *C* is one more than prime root *p*, then $U_i(p-1)$ is assigned a value zero and $U_i(p)$ is assigned a value *p*. In this case, if the size of input bits *K* is equal to the interleaver matrix size $R \times C$, then the first and last bits of the last row are exchanged, i.e., $U_{R-1}(p)$ with $U_{R-1}(0)$.

7. The inter-row permutation is performed for the intra-row permuted rectangular matrix based on the permutation pattern T introduced in step 4.

4.1.1.3 Bits output from the rectangular matrix with pruning

After intra-row and inter-row permutations, data is read out from the permuted $R \times C$ rectangular matrix in a columnwise fashion starting with the bit in row 0 of column 0 and ending with the bit at row *R-1* of column *C-1*.

$$\begin{bmatrix} Y'_1 & Y'_{(R+1)} & Y'_{(2R+1)} & \dots\dots & Y'_{((C-1)R+1)} \\ Y'_2 & Y'_{(R+2)} & Y'_{(2R+2)} & \dots\dots & Y'_{((C-1)R+2)} \\ \vdots & \vdots & \vdots & & \vdots \\ Y'_R & Y'_{2R} & Y'_{3R} & \dots\dots & Y'_{C\times R} \end{bmatrix}$$

The output is pruned by deleting dummy bits that were added before interleaving. For example, bits Y'_k that correspond to bits Y_k for $k > K$ were the dummy bits and are removed from the output. The bit sequence obtained after interleaving and pruning is X'_k for $k \leq K$ where X'_1 corresponds to Y'_k with smallest index k after pruning. The number of bits output from the turbo code interleaver is equal to the number of input bits *K* and the number of pruned bits is $(R \times C)-K$.

4.2 Channel

After encoding, the entire *n* bit turbo codeword is assembled into a frame, modulated, transmitted over the channel, and then decoded. The input to the modulator is assumed to be U_k where U_k can be either systematic bit or parity bit and it can have a value either

0 or 1. The signal passing received after passing through the channel and demodulation is Y_k which can take on any value. Thus, U_k is a hard value an Y_k is a soft value.

The modulation scheme assumed for the proposed system was BPSK and the channel model can be either AWGN or flat fading channel. For a channel with channel gain a_k and Gaussian noise n_k, the output from the receiver's matched filter is $Y_k = a_k S_k + n_k$ where S_k is the signal after passing through the matched filter. For systematic data bits, $S_k = 2X_k - 1$. For upper and lower encoder parity bits $S_k = 2Z_k - 1$ and $S_k = 2Z_k' - 1$ respectively. For an AWGN channel, channel gain a_k equals one whereas for a Rayleigh flat fading channel, it is a Rayleigh random variable. The Gaussian noise n_k has a variance $\sigma^2 = 1/(2E_s / N_o)$ where E_s is the energy per code bit and N_o is the one sided noise power spectral density. For a coderate $r = K/(3K+12)$, the noise variance in terms of energy per data bit E_b becomes

$$\sigma^2 = \frac{1}{(2r\,E_b/N_o)} = \frac{(3K+12)}{\left(2K(E_b/N_o)\right)} \tag{4.12}$$

The input to the decoder is in the log likelihood ratio LLR form to assure that the effects of the channel .i.e. noise variance n_k and channel gain a_k have been properly taken into account. The input to the decoder is in the form

$$R_k = ln\left(\frac{P[S_k = +1|\,Y_k]}{P[S_k = -1|\,Y_k]}\right) \tag{4.13}$$

By applying Bayes's rule and assuming that $P[S_k=+1] = P[S_k=-1]$

$$R_k = ln((P[S_k = +1 \cap Y_k]/P[Y_k])/(P[S_k = -1 \cap Y_k]/P[Y_k]))$$

$$= ln(P[S_k = +1 \cap Y_k]/P[S_k = -1 \cap Y_k])$$

$$= ln((P[\,Y_k|\,S_k = +1] \times P[S_k = +1])/(P[\,Y_k|\,S_k = -1] \times P[S_k = -1]))$$

$$= ln((P[\,Y_k|\,S_k = +1]\,)/(P[\,Y_k|\,S_k = -1]\,))$$

$$R_k = ln\left(\left(f_y[\,Y_k|\,S_k = +1]\,\right)/\left(f_y[\,Y_k|\,S_k = -1]\,\right)\right) \tag{4.14}$$

where $f_Y(Y_k|S_k)$ is the conditional pdf of receiving Y_k given code bit S_k, which is Gaussian with mean a_kS_k and variance σ^2 [27]. Thus, for decoding the bit, we need both the code bit as well as knowledge of the statistics of the channel.

Substituting the expression for Gaussian pdf and simplifying yields:

$$R_k = \frac{2\,a_k}{\sigma^2}\,Y_k \tag{4.15}$$

The expression shows that the matched filter coefficients need to be scaled by a factor $2a_k/\sigma^2$ in order to meet the input requirements of the decoder where R_k shows the received LLR for both data and parity bits. The notation $R(X_k)$ denotes the received LLR corresponding to systematic data bits X_k, $R(Z_k)$ is the LLR corresponding to the upper encoder parity bit Z_k and $R(Z_k')$ corresponds to the lower encoder parity bit Z_k'.

4.3 Decoder architecture

The architecture of the UMTS decoder is as shown in Figure 4.2. It operates in an iterative manner as indicated by the presence of the feedback path [26] [27].

The decoder uses the received codeword along with the knowledge of the code structure to compute the LLRs. Because of the presence of the interleaver at the encoder end, the structure of the code becomes very complicated and it is not feasible to compute the LLR by using a single probabilistic processor. It is rather feasible to break the job of achieving a global LLR estimate in two iterations. As a result, each iteration of the Turbo decoder consists of two half iterations, one for each constituent RSC decoder. As indicated by the sequence of arrows, the timing of the decoder is such that the RSC decoder 1 operates during the first half iteration and RSC decoder 2 operates during the second half iteration.

Both the decoder structures compute LLR using a soft-input-soft-output processor (SISO). Although the data bits used by the second decoder are interleaved, the two SISO processors are producing LLR estimate of same set of data bits in different order.

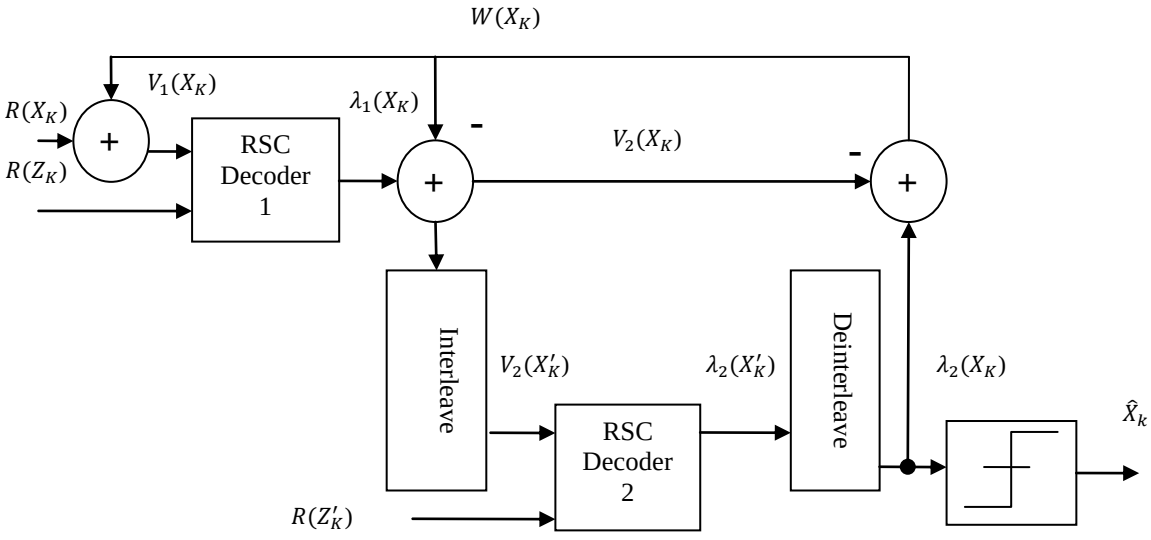


Figure 4.2 The UMTS turbo decoder architecture [26].

The performance of the decoders can be greatly improved by sharing the LLR estimates with each other. The first SISO decoder calculates the LLR and passes it to the second decoder. The second decoder uses that value to calculate LLR which is fed back to the first decoder. These back and forth exchange of information between the processors make turbo decoder an iterative decoder. This also results in an improved performance as after every iteration, the decoder is better able to estimate the data. However as the number of iterations increase, the rate of performance improvements decreases [27].

The value $w(X_k)$ for $1 \leq k \leq K$, is the extrinsic information produced by RSC decoder 2 and fed back to the input of RSC decoder 1. Prior to the first iterations, the value of $w(X_k)$ is initialized to all zeros as decoder 2 has not yet decoded the data. After each iteration, the value of $w(X_k)$ is updated to a new non zero value to reflect the beliefs regarding the data and are propagated from decoder 2 to decoder 1 which uses this information as the extrinsic information. As both RSC encoders at the transmitter end are using independent tail bits so only the information regarding actual information bits will be exchanged between the two encoders. For the value of k in range $K+1 \leq k \leq K+3$, $w(X_k)$ is undefined or in other words zero after every iteration.

Decoder 1 must use the extrinsic information while decoding. Figure 4.2 gives a detailed understanding of the steps being followed. The following sequence of steps is followed:

1. The extrinsic information $w(X_k)$ is added to the received systematic LLR $R(X_k)$ forming a new variable $V_1(X_k)$.
   $$V_1(X_k) = R(X_k) + w(X_k) \tag{4.16}$$
   As the extrinsic information $w(X_k)$ is non zero for $1 \leq k \leq K$ so the input to the RSC decoder 1 is the received parity bits in LLR form $R(Z_k)$ and a combination of the systematic data $R(X_k)$ and extrinsic information $w(X_k)$, i.e., $V_1(X_k)$. For $K+1 \leq k \leq K+3$ (tail bits) no extrinsic information is available, i.e., $w(X_k)$ equals zero. In this case, the input to RSC decoder 1 is the received and scaled upper encoder's tail bits $V_1(X_k) = R(X_k) + 0 = R(X_k)$, and the corresponding received and scaled parity bits $R(Z_k)$.
2. The RSC decoder 1 uses this information to decode the data and outputs the LLR $\Lambda_1(X_k)$ for $1 \leq k \leq K$ as only the LLR of data bits is shared with the second encoder.
   $$\begin{aligned}\Lambda_1(X_k) &= RSC\ decode\ [R(Z_k), V_1(X_k)] \\ &= RSC\ decode\ [R(Z_k), \big(R(X_k) + w(X_k)\big)]\end{aligned} \tag{4.17}$$
3. The extrinsic information $w(X_k)$ is subtracted from the first encoder's LLR $\Lambda_1(X_k)$, to obtain a new variable $V_2(X_k)$ which is to be used by the second encoder. Similar to $V_1(X_k)$, $V_2(X_k)$ contains both the systematic channel LLR and the extrinsic information produced by decoder 1.

$$V_2(X_k) = \Lambda_1(X_k) - w(X_k) \tag{4.18}$$

4. The sequence $V_2(X_k)$ is interleaved to obtain $V_2(X_k')$ which is input to decoder 2.
$$V_2(X_K') = Interleave\left(V_2(X_k)\right) \tag{4.19}$$
This is done so that the sequence of bits in $V_2(X_k')$ matches that of the input to the second decoder R(Z_k'), the bits which were bits interleaved before convolutional coding.
5. For 1 ≤ k ≤ K, the input to the decoder is $V_2(X_k')$, i.e., the interleaved version of $V_2(X_k)$ and $R(Z_k')$, the channel's LLR corresponding to second encoder parity bits. The decoder uses this input to generate the LLR $\Lambda_2(X_k{}')$.
$$\begin{aligned}\Lambda_2(X_k') &= RSC\ decode\left[R(Z_k'), V_2(X_k')\right] \\ &= \left[R(Z_k'), Interleave\left(V_2(X_k)\right)\right] \\ &= \left[R(Z_k'), Interleave\left(\Lambda_1(X_k) - w(X_k)\right)\right]\end{aligned} \tag{4.20}$$

6. The LLR $\Lambda_2(X_k')$ is deinterleaved to form $\Lambda_2(X_k)$ with the sequence of bits now same as the original bits.
$$\Lambda_2(X_k) = Deinterleave\left(\Lambda_2(X_k')\right) \tag{4.21}$$
7. The sequence $\Lambda_2(X_k)$ is fed back to obtain the extrinsic information $w(X_k)$ by subtracting $V_2(X_k)$ from it. This value of $w(X_k)$ will be used by decoder 1 during the next iteration.
$$w(X_k) = \Lambda_2(X_k) - V_2(X_k) \tag{4.22}$$
8. When the number of iterations are completed, a hard bit decision is taken using $\Lambda_2(X_k)$ to obtain the decoded bit $\hat{X}_k$ such that $\hat{X}_k$ equals one when $\Lambda_2(X_k)$ is greater than zero and vice versa for $1 \le k \le K$.
$$\hat{X}_k = 1 \quad if \quad \Lambda_2(X_k) > 0 \tag{4.23}$$
$$\hat{X}_k = 0 \quad if \quad \Lambda_2(X_k) \le 0$$

4.4 RSC decoder

The heart of the turbo decoder is the algorithm used to implement the RSC decoder. The RSC decoders use a trellis diagram to represent all possible transitions between the states along with their respective outputs. As the RSC encoder used by UMTS has three shift registers, the number of distinct possible states is 2^3, i.e., eight. The trellis diagram

not only shows the state for a particular bit k but also shows the permissible state transitions leading to next state [26][27].

While decoding, each of the two RSC decoders sweep through the code trellis twice, once in the forward and once in the backward direction. The sweep is implemented using MAP algorithm which is a modified version of the Viterbi algorithm to compute partial branch and path metrics. Although it is somewhat more complex than the Viterbi algorithm but it has the advantage that it outputs the APPs of the information and the channel [29]. The presented algorithm uses a version of the classical MAP algorithm in the log domain.

4.4.1 Max* operator

The MAP algorithm poses technical difficulties because of a high number of additions and multiplications. MAP algorithm if implemented in the log domain [30] [31] significantly reduces the computational complexity. The log-MAP algorithm is based on the Viterbi algorithm with two key modifications [31]:

- ✓ Trellis should be swept in both forward and reverse directions.
- ✓ Jacobi algorithm also known as max* operator should be used instead of the add-compare-select (ACS) operation of the Viterbi algorithm.

The log-MAP algorithm is implemented twice during each half iteration; once in the forward and once in reverse direction. It constitutes a dominant portion of the decoder complexity and thus, the manner in which it is implemented is critical in determining the performance and complexity of the decoder. Four versions based on four different max* operations have been considered in the implementation [26].

4.4.1.1 Log MAP algorithm

The logarithm version of the MAP algorithm is the log MAP [31] algorithm. It has reduced complexity as multiplication operations are transformed into additions.

$$\begin{aligned} max^*(x,y) &= ln(e^x + e^y) \\ &= max(x,y) + ln\left(1 + e^{-|y-x|}\right) \\ &= max(x,y) + f_c(|y-x|) \end{aligned} \tag{4.24}$$

Thus, max* operator in this case be calculated by taking the maximum of the two input arguments x and y and then adding a correction function f_c to it. The correction function is simply a function of the absolute difference between the two arguments of the max* operator. The correction function f_c can be computed using log and exponential functions or it can be pre-computed and stored in a look-up table to decrease

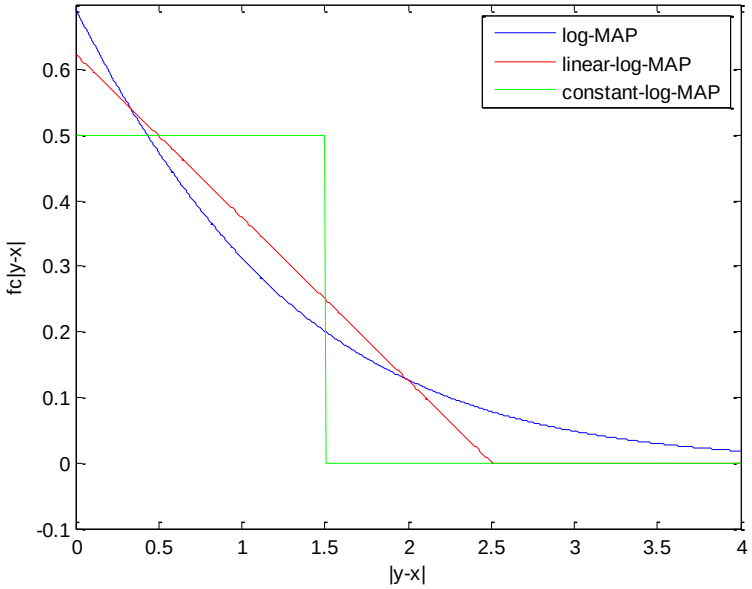


Figure 4.3 The correction function fc used by log-MAP, constant-log-MAP and linear-log-MAP algorithm.

complexity. The log-MAP algorithm is the most complex of all the four algorithms but it offers the best BER performance. The correction function f_c used by log MAP algorithm is illustrated in Figure 4.3 along with the correction functions used by constant-log MAP and linear-log MAP algorithms.

4.4.1.2 Max-log MAP algorithm

In the max-log MAP algorithm, the log MAP algorithm is loosely approximated by setting the correction function f_c to zero in equation (4.24), i.e., it is not used at all [27].

$$max^*(x, y) \approx \max(x, y) \tag{4.25}$$

The max-log MAP algorithm is the least complex of all the four algorithms but it has still twice the complexity of the Viterbi algorithm. It can be implemented using a pair of Viterbi algorithm, one that sweeps through the trellis in the forward direction and one in the reverse direction [31]. It offers the worst BER performance of all the four variants of MAP algorithm. It has however the additional benefit of being intolerant of imperfect noise variance estimates while operating in an AWGN channel.

4.4.1.3 Constant-log MAP algorithm

The constant-log MAP algorithm was first introduced by W. J. Gross and P. G. Gulak in 1998 [33]. It uses a lookup table with only two entries of the correction function, thus decreasing the complexity.

$$\max^{*}(x,y) \approx \max(x,y) + \begin{cases} 0 \text{ if } |y-x| > T \\ C \text{ if } |y-x| \leq T \end{cases} \tag{4.26}$$

The values of *T* and *C* used in the implementation are 1.5 and 0.5 respectively [26] [34]. The performance and complexity of constant-log MAP algorithm lies between log MAP and max-log MAP however, it is more susceptible to noise variance estimation errors than log MAP.

4.4.1.4 Linear –log MAP algorithm

In the constant-log MAP algorithm, the correction function was approximated by a set of lookup tables. This implies the need of a high speed memory. To avoid the need of high-speed memory, a linear approximation can be used. The linear-log MAP algorithm is also based on using a linear approximation of the Jacobi algorithm [35].

$$max^{*}(x,y) \approx \max(x,y) + \begin{cases} 0 & if\ |y-x| < T \\ a(|y-x| - T) & if\ |y-x| \geq T \end{cases} \tag{4.27}$$

For a floating point processor, the parameters a and T used by the linear approximation are chosen to minimize the total squared error between the exact correction function and its linear approximation [26].

$$\varphi(a,T) = \int_0^t [a(x-T) - ln(1+e^{-x})]^2\, dx + \int_T^{\infty} [ln(1+e^{-x})]^2\, dx \tag{4.28}$$

To minimize the function, partial derivates w.r.t a and T are set to zero. The partial derivated of equation (4.26) w.r.t a is

$$\begin{aligned} \frac{\partial\varphi(a,T)}{\partial a} &= \int_0^T 2[a(x-T) - ln(1+e^{-x})](x-T)dx \\ &= 2\int_0^T a(x-T)^2 dx - 2\int_0^T (x-T)\ ln(1+e^{-x})\, dx \\ &= \frac{2}{3}a(x-T)^3\Big|_0^T - 2\int_0^T (x-T)\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\, e^{-nx}\, dx \\ &= \frac{2}{3}a[(T-T)^3 - (0-T)^3] - 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\int_0^T (x-T)e^{-nx}dx \\ &= \frac{2}{3}aT^3 + 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\left[\frac{1}{n}\left\{T + \left(\frac{1}{n}\right)(e^{-nT}-1)\right\}\right] \end{aligned}$$

$$= \frac{2}{3}aT^3 + 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}\,T + 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}\,(e^{-nT}) - 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}$$

$$= \frac{2}{3}aT^3 + \ 2K_1T - 2K_2 + 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}e^{-nT} \tag{4.29}$$

Where K_1 and K_2 are constants :

$$K_1 = \sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}$$

$$K_2 = \sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}$$

Now taking partial derivative of $\varphi(a,T)$ of equation (4.28) with respect to T

$$\begin{aligned}
\frac{\partial\varphi(a,T)}{\partial T} &= \int_0^T 2[a(x-T) - ln(1+e^{-x})]\,(-a)dx \\
&\quad + \ [a(x-T) - ln(1+e^{-x})]^2|_{x=T} - [ln(1+e^{-x})]^2|_{x=T} \\
&= \int_0^T 2[a(x-T) - ln(1+e^{-x})](-a)x \\
&= \int_0^T [a(x-T) - ln(1+e^{-x})]dx \\
&= \int_0^T a(x-T)dx - \int_0^T ln(1+e^{-x})\ dx \\
&= \frac{a}{2}(x-T)^2\Big|_0^T - \int_0^T \sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}e^{-nx}dx \\
&= \frac{-aT^2}{2} - \sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\int_0^T e^{-nx}dx \\
&= \frac{-aT^2}{2} - \sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}[e^{-nT} - 1] \\
\frac{\partial\varphi(a,T)}{\partial T} &= -\frac{aT^2}{2} - K_1 + \sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}e^{-nT}
\end{aligned} \tag{4.30}$$

Adding 3/2 times equation (4.29) and *2T* times equation (4.30) yields

$$g(T) = \left(aT^3 + \ 3K_1T - 3K_2 + 3\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}e^{-nT}\right)$$

$$+(-aT^3 - 2TK_1 + 2T\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}e^{-nT})$$

$$g(T) = K_1T + 2T + 3\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}e^{-nT} + 2T\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}e^{-nT} \quad (4.31)$$

By solving *g(T)* which is momotonically increasing function of *T*, we arrive at the optimal value of *T*. An iterative approach is used by first choosing T_1 and T_2 such that $g(T_1) < 0$ and $g(T_2) > 0$. The mid point between T_1 and T_2 is T_0. If $g(T_0) < 0$, T_1 is set equal to T_0, otherwise T_2 is set equal to T_0. The iterations are repeate until T_1 ans T_2 become very close. In the implementaion, the upper limit of the summation is set to 30 as with this value an error of 10^{-10} can be achieved. By iteratively solving equation (4.29) for *g(T)* and setting the upper value of summation to 30, the value of *T* and a come out to be 2.50681640022001 and -0.24904181891710 respectively. For the simulation, the value of *T* and *a* are approximated to 2.5068 and -0.24904 respectively. The complexity and performance of linear-log-MAP algorithm lies between that of log-MAP and constant log-MAP algorithms however it converges much faster than the constant-log-MAP algorithm.

4.4.2 RSC decoder operation

As discussed earlier, each of the two RSC decoders sweep through the trellis twice, once in forward and once in reverse direction. However, it does not matter in which direction the sweep is performed first, i.e., one can sweep in either the forward or the reverse direction first. Also, the partial path metrics for only the entire first sweep must be stored in memory. The partial path metrics for the entire second sweep need not be stored as the LLR values can be computed during the second sweep. Thus, partial path metrics for only the current state and the previous state must be stored during the second sweep. It is recommended to perform the reverse sweep first and save partial path metrices for each node in memory. Then the forward sweep is performed and LLR estimates of data are produced. If the forward sweep is performed first, the LLR estimates are produced during the reverse sweep and are thus in reverse order [26].

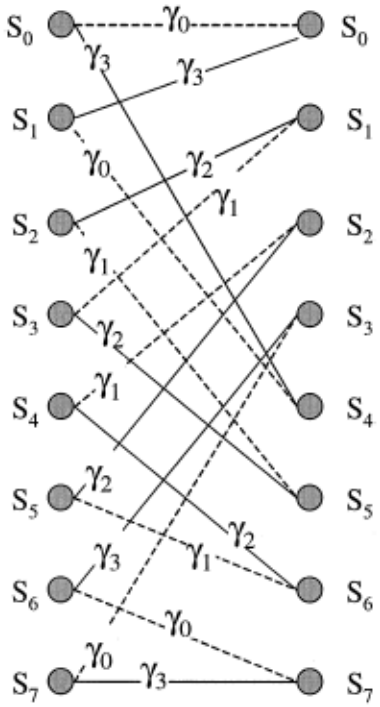


Figure 4.4 A trellis section for the RSC code used by the UMTS turbo code [26].

The trellis structure used by the RSC decoder is shown in Figure 4.4. Each state has two branches leaving it, one corresponding to an input one and one for input zero. Solid lines indicate data one and dotted lines indicate data zero. The branches indicate which next state can be reached from a particular state. The branches are labelled with branch metrics $\boldsymbol{\gamma}$. Every distinct codeword follows a particular path in the trellis [27] [26]

The branch metric connecting state S_i (previous state, on left) and state S_j (present state, on right) is denoted as γ_{ij}. The branch metric depends on the data bit $X(i,j)$ as well as the parity bit $Z(i,j)$ associated with the branch. The branch metric is given as:

$$\gamma_{ij} = V(X_k)X(i,j) + R(Z_k)Z(i,j) \tag{4.32}$$

The RSC encoder being rate *r=1/2*, only four distinct branch metrics are possible:

$$\begin{aligned}
&\gamma_0 = 0 && X(i,j) = 0, \quad Z(i,j) = 0 \\
&\gamma_1 = V(X_k) && X(i,j) = 1, \quad Z(i,j) = 0 \\
&\gamma_2 = R(Z_k) && X(i,j) = 0, \quad Z(i,j) = 1 \\
&\gamma_1 = V(X_k) + R(Z_k) && X(i,j) = 1, \quad Z(i,j) = 1
\end{aligned} \tag{4.33}$$

where $V(X_k)$ and $R(Z_k)$ are the inputs to the RSC decoder. In Figure 4.2, for the first (upper) RSC decoder, $V(X_k)$ equals $V_1(X_k)$ and for the second (lower) decoder, $V(X_k)$ equals $V_2(X_k^{'})$. Also for the lower decoder, the second input is the parity bits corresponding to the second encoder, i.e., $R(Z_k')$ instead of $R(Z_k)$.

4.4.2.1 Backward recursion

The simulated decoder sweep through the trellis in the backward direction first. The normalized partial path metrics at all the nodes in the trellis are saved in memory. These normalized partial metrics denoted by β are later used to calculate LLRs during the forward recursion [26]. The partial path metrics at trellis stage *k* with $2 \leq k \leq K+3$ and at state S_i for $(0 \leq i \leq 7)$ is denoted as $\beta_k(S_i)$. The backward recursion starts at stage $K+2$ and proceeds through the trellis in the backward direction until stage 2 is reached. The recursion is initialized with $\beta_{K+3}(S_0)$ equal to zero and rest all states initialized to negative infinity.

$$\beta_{K+3}(S_0) = 0, \quad \beta_{K+3}(S_i) = -\infty \quad \forall i > 0 \tag{4.34}$$

The partial path metrics for the current state *k* are found using the partial path metrics for the next (previously calculated) $k+1$ state and the associated branch metrics γ_{ij}.

$$\tilde{\beta}_k(S_i) = max^*\{(\beta_{k+1}(S_{j_1}) + \gamma_{ij_1}), (\beta_{k+1}(S_{j_2}) + \gamma_{ij_2})\} \tag{4.35}$$

Where $\tilde{\beta}_k(S_i)$ indicates the unnormalized partial path metric. The states S_{j_1} and S_{j_2} indicate the two states at stage $k+1$ that are connected to state S_i in trellis at stage *k*. The partial path metrics are normalized after the calculation of $\tilde{\beta}_k(S_0)$ to obtain normalized partial path metrics.

$$\beta_k(S_i) = \tilde{\beta}_k(S_i) - \tilde{\beta}_k(S_0) \tag{4.36}$$

The purpose of normalization is to reduce memory requirements. As after normalization $\beta_k(S_0)$ equals zero, so only the other seven normalized partial path metrics $\beta_k(S_i)$ for $1 \leq i \leq 7$, need to be stored. As a result, there is a 12.5% saving in memory compared to when no normalization was used.

4.4.2.2 Forward recursion and LLR calculation

Once the backward recursion is complete, the trellis is then swept in the forward direction. Unlike the backward recursion where the partial path metrics for all states at all stages need to be stored in memory, for the forward recursion only the partial path metrics for two stages of the trellis need to be stored in memory, i.e., the current stage *k* and the previous stage $k-1$. The forward partial path metric for state S_i at trellis stage *k* is denoted as $\alpha_k(S_i)$ with the stages *k* ranging from 0 to $K-1$ and all possible states $0 \leq i \leq 7$. The forward recursion is initialized by setting the value of forward partial path metric for stage zero, α_0 initialized to zero for state S_0 and negative infinity for all other seven states.

$$\alpha_0(S_0) = 0, \ \alpha_0(S_i) = -\infty \quad \forall i > 0 \tag{4.37}$$

The forward recursion begins with stage *k=1* and the trellis is swept in the forward direction until stage *K* is reached. The un-normalized partial path metrics $\tilde{\alpha}_k(S_i)$ are calculated as

$$\tilde{\alpha}_k(S_i) = max^*\{(\alpha_{k-1}(S_{j_1}) + \gamma_{i_1 j}), (\alpha_{k-1}(S_{j_2}) + \gamma_{i_2 j})\} \tag{4.38}$$

Where S_{j_1} and S_{j_2} are the two states at stage *k-1* that are connected to state S_j at stage *k*. The partial path metrics $\tilde{\alpha}_k(S_i)$ are normalized to $\alpha_k(S_i)$ after the calculation of $\tilde{\alpha}_k(S_0)$

$$\alpha_k(S_i) = \tilde{\alpha}_k(S_i) - \tilde{\alpha}_k(S_0) \tag{4.39}$$

Using these values of normalized partial path metrics $\alpha_k(S_i)$ are computed for stage *k* along with the LLR estimate for data bit X_k. For LLR calculation, first the likelihood of the branch connecting state S_i at stage *k-1* to state S_j at stage *k* is calculated. It is denoted by $\lambda_k(i,j)$.

$$\lambda_k(i,j) = \alpha_{k-1}(S_i) + \gamma_{ij} + \beta_k(S_j) \tag{4.40}$$

The likelihood of data 1 or 0 is then the Jacobi algorithm of the likelihood of all the branches corresponding to data 1 or zero.

$$\Lambda(X_k) = \max^*_{(S_i \to S_j): X_i = 1} \{\lambda_k(i,j)\} - \max^*_{(S_i \to S_j): X_i = 0} \{\lambda_k(i,j)\} \tag{4.41}$$

The max^* operator is computed recursively over the over the likelihoods of all the data one branches $\{(S_i \to S_j): X_i = 1\}$ or data zero branches $\{(S_i \to S_j): X_i = 0\}$. Once $\Lambda(X_k)$ is calculated, $\alpha_{k-1}(S_i)$ is no longer needed and may be discarded.

4.5 Stopping criteria for UMTS turbo decoders

The complexity of the decoder increases with the increase in the number of iterations whereas the performance of the decoder improves. There is always a trade-off between the performance of the decoder and its complexity. Standard turbo code implementations use a fixed number of iterations specified before starting the coding process. However, the computational complexity can be decreased by keeping the number of iterations variable [37].

Different iteration control criterions are available for Turbo codes. The code keeps on iterating until a certain rule or stopping condition is satisfied. The stopping condition is computed based on the information available during decoding and it determines when the iteration process will be terminated. At the time of termination, the data block is either successfully decoded or it cannot be decoded at all. After every iteration, the decoder checks the stopping condition. If true, the code terminates. Otherwise it continues to the next iteration [37].

The main code termination criteria used in the simulation is the average mutual information of LLR after each iteration. If the mutual information measurement has worsened in the iteration as compared to the previous iteration, the code will terminate early [38].

5. MATLAB IMPLEMENTATION AND ANALYSIS OF RESULTS

The UMTS Turbo code was explained in detail in Chapter 4. For the thesis, the turbo code was implemented in MATLAB.

5.1 MATLAB implementation

The input data frame size is between 40 and 5114 bits as is the size of the UMTS interleaver. The turbo encoder consists of two main blocks, i.e., the recursive convolutional encoder and the interleaver. The convolutional encoded data is arranged in the sequence described in equations (4.4) and (4.5). The encoded data frame is BPSK modulated and sent over the channel. The channel models used in the simulation are AWGN and Rayleigh flat fading channel. After adding noise to the data, the LLR is calculated as in equation (4.15). The control flow graph of the implemented code is shown in Figure 5.1.

The decoder implementation is complex and computationally extensive. It includes processing using a number of loops. A limit is set on the maximum number of bits to be encoded and maximum allowable error for early termination of the code. The decoder decodes iteratively checking the number of errors after every iteration. If the number of errors is zero for an iteration, the code will not execute the next iteration to decrease processing load. If the error exists, the average mutual information is checked. If there is an improvement, the code goes into next iteration. If the mutual information average fails to improve for a specified number of times, the decoder is terminated at that iteration. It then goes back to the loop and checks if the number of data bits to be encoded has reached the maximum limit or error is less than maximum allowable error. If true, it repeats the steps described above. Otherwise, it increments the SNR and performs the encoding and decoding [36] [37] [34].

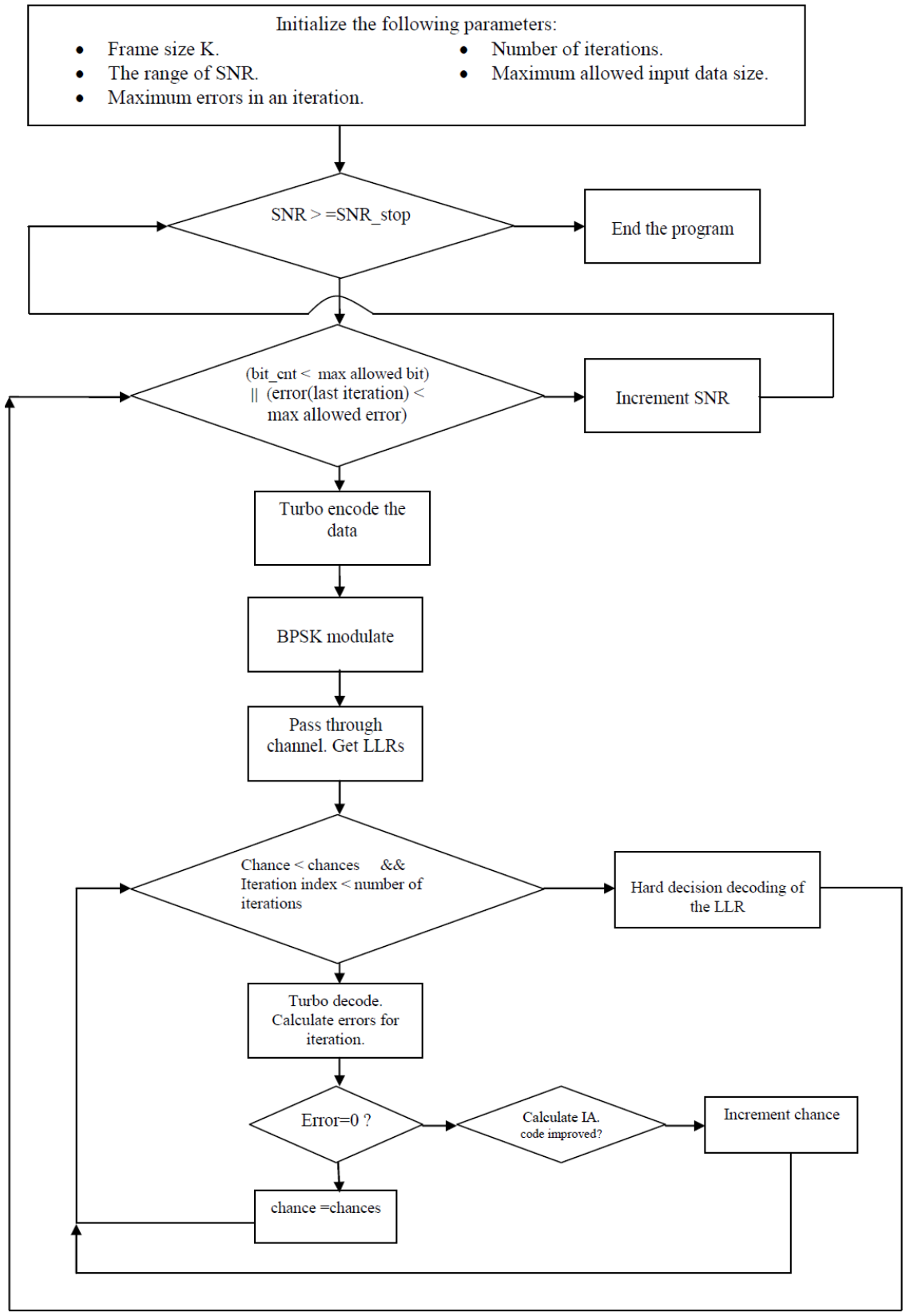


Figure 5.1 Control flow chart of UMTS turbo code MATLAB implementation.

5.2 Turbo code error performance analysis

Turbo codes are capable of producing low error rates at astonishingly low SNRs if the frame size K is kept large and permutations are selected randomly. Turbo codes are iterative codes and the exchange of mutual information between the constituent decoders results in better performance of the code. The more is the number of iterations; more is the exchange of information between the constituent encoders. As a result, the code can perform better. However, when the number of iterations is made too high, there is not much improvement in performance. Thus, upper bounds need to be specified for the number of iterations and SNR for optimal performance of the code.

As the turbo code decoding is computationally very intensive so most of the simulated performance results are for small frame lengths.

5.2.1 BER performance for frame size K = 40

The turbo code program was simulated for frame size K = 40 over a Rayleigh fading channel. To keep the simulation fast, the number of frames for each SNR was taken as 500. Thus, for a frame size 40, 20000 bits were sent at each SNR value to get the BER. The SNR range was used from 0 to 5 dB. The number of decoder iterations was chosen to be 10. The BER for the iterations is shown in Figure 5.2.

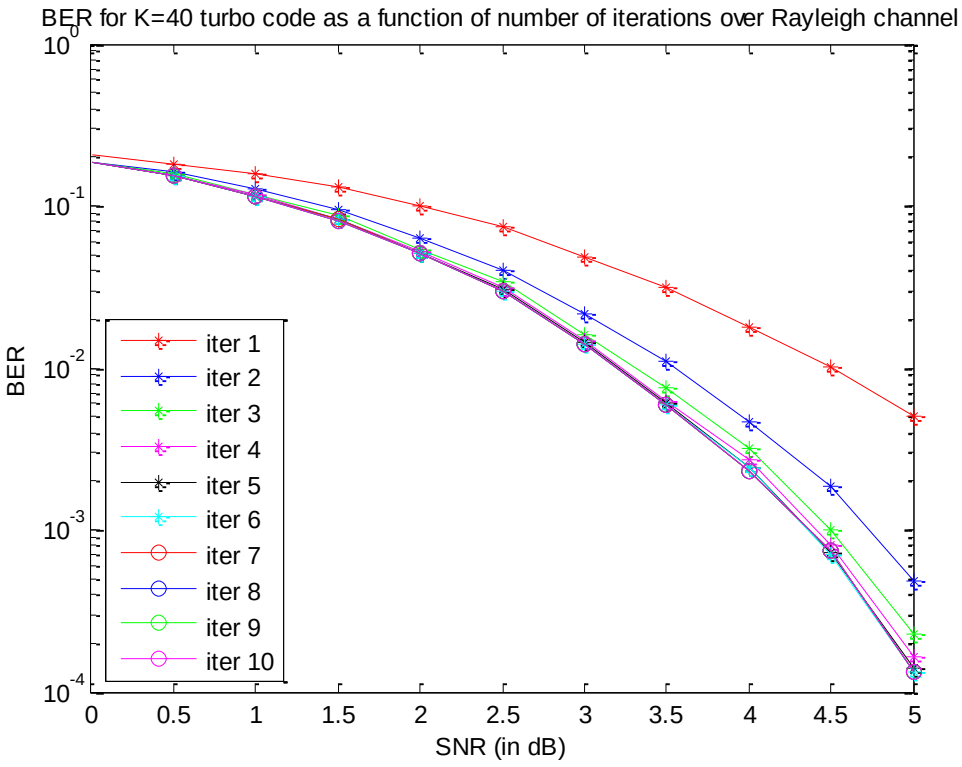


Figure 5.2 BER for frame size K = 40 UMTS turbo code over Rayleigh channel.

Table 5.1 BER for K = 40 UMTS turbo code over Rayleigh channel.

Iteration	BER	Iteration	BER
1	0.005	6	1.338×10^{-4}
2	4.886×10^{-4}	7	1.339×10^{-4}
3	2.285×10^{-4}	8	1.339×10^{-4}
4	1.655×10^{-4}	9	1.34×10^{-4}
5	1.418×10^{-4}	10	1.34×10^{-4}

The BER values at the end of each iteration are given in Table 5.1 BER for K = 40 UMTS turbo code over Rayleigh channel. The turbo code was able to achieve BER of a 0.5×10^{-2} after first decoder iteration. The BER improved to 1.34×10^{-4} after the 10^{th} iteration. It can be seen that as the number of iteration increases, the BER performance improves. However, the rate of improvement decreases. This is depicted by the overlapping curves after 5^{th} iterations. The BER after 5 iterations is 1.418×10^{-4}. The BER does not show significant improvement after 5^{th} iteration. Thus, the number of iterations should be kept such as to avoid extra computations.

The BER curve for frame size K = 40 over an AWGN channel is shown Figure 5.3 BER for frame size K = 40 UMTS turbo code over an AWGN channel. The BER after the first decoder iteration is 3.628×10^{-4}. The BER decreases with the increase of iterations and at the end of tenth iteration; the BER is 1.152×10^{-5}.

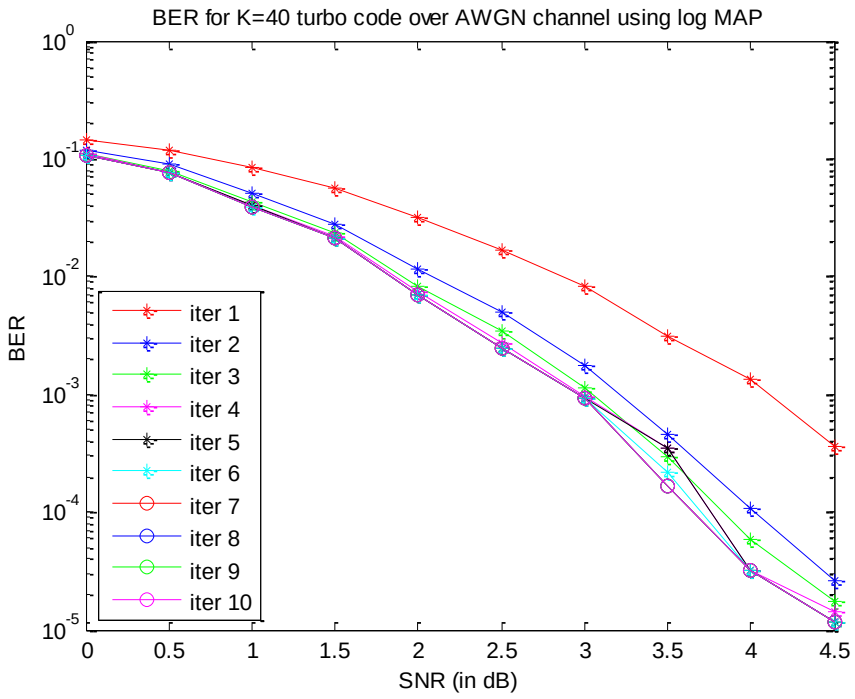


Figure 5.3 BER for frame size K = 40 UMTS turbo code over an AWGN channel.

5.2.2 BER performance for frame size K = 100

Turbo code performance can be improved by increasing the frame size K. The code can achieve higher BER with the increase of frame size. This is because the interleaver permutes the data and the decoder is better able to decode the data. The simulation for the turbo code was run with frame size K = 100 keeping the number of iterations 6. The BER for the iterations is given in Table 5.2.

The BER curve for frame size K = 100 UMTS turbo code is shown in Figure 5.4. The figure shows an improvement in the BER performance as compared to the frame size K 40. The BER decreases with the increase in the number of iterations. This behaviour is similar to the case with frame size 40. However, the frame size K 100 is able to achieve BER of almost 10^{-4} at SNR of 2.5 dB. The frame size K 40 was able to achieve such BER at SNR 5 dB. Thus, we can conclude that by increasing the frame size K, the code can achieve the same BER at much lower SNR.

Table 5.2 BER for frame size K = 100 UMTS turbo code.

Iteration	BER	Iteration	BER
1	1.07×10^{-2}	4	1.472×10^{-4}
2	5.662×10^{-4}	5	1.245×10^{-4}
3	2.1518×10^{-4}	6	1.246×10^{-4}

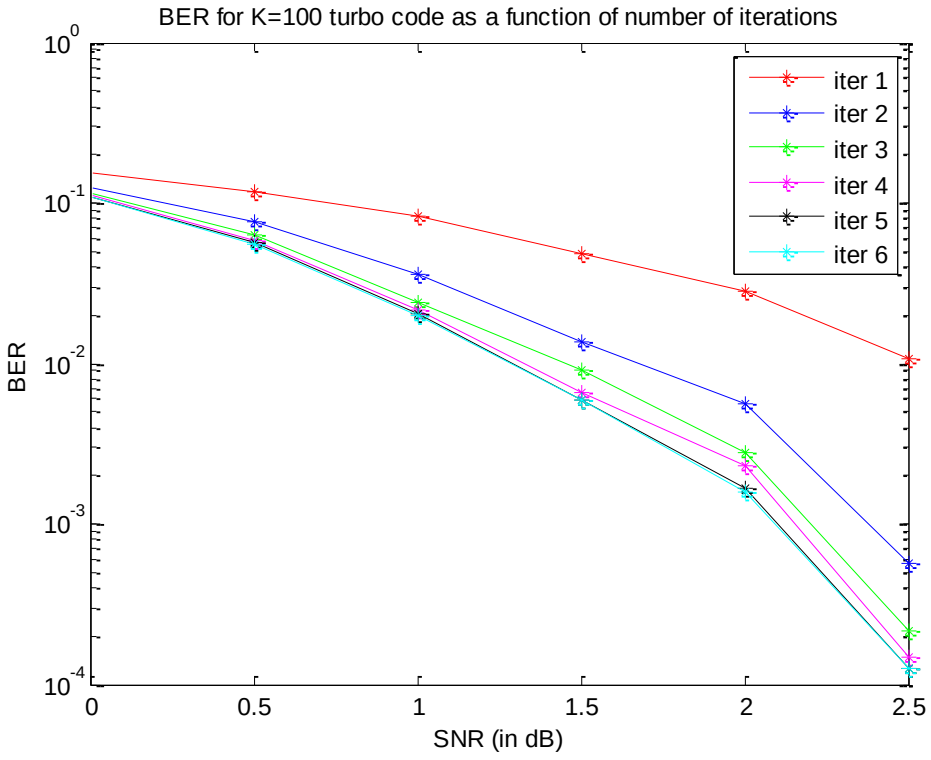


Figure 5.4 BER for frame size K = 100 UMTS turbo code after 6 decoder iterations.

5.2.3 BER as a function of frame size K (Latency)

The performance comparison of turbo code can be done by plotting the BER for different frame sizes K as in Figure 5.5. The figure shows that by increasing the frame size K, the BER performance of the code improves. As a result, lower BER can be achieved by keeping the SNR constant. The BER values for frame size K 40, 100 and 320 is given in Table 5.3. It can be seen in the figure that for frame size K 320, BER of 1.75×10^{-5} is achieved at SNR 1.5 dB. However, frame size K 100 and 40 achieve only a BER of 0.76×10^{-2} and 0.0212 respectively for the same SNR. Larger frame sizes mean more latency as the encoding and decoding is done per frame. Thus, the performance improvement is achieved at the cost of increased latency.

Table 5.3 BER for frame size K = 40, 100 and 320.

Frame size K	BER	SNR (dB)
40	1.152×10^{-5}	4.5
100	7.745×10^{-6}	3
320	1.75×10^{-5}	1.5

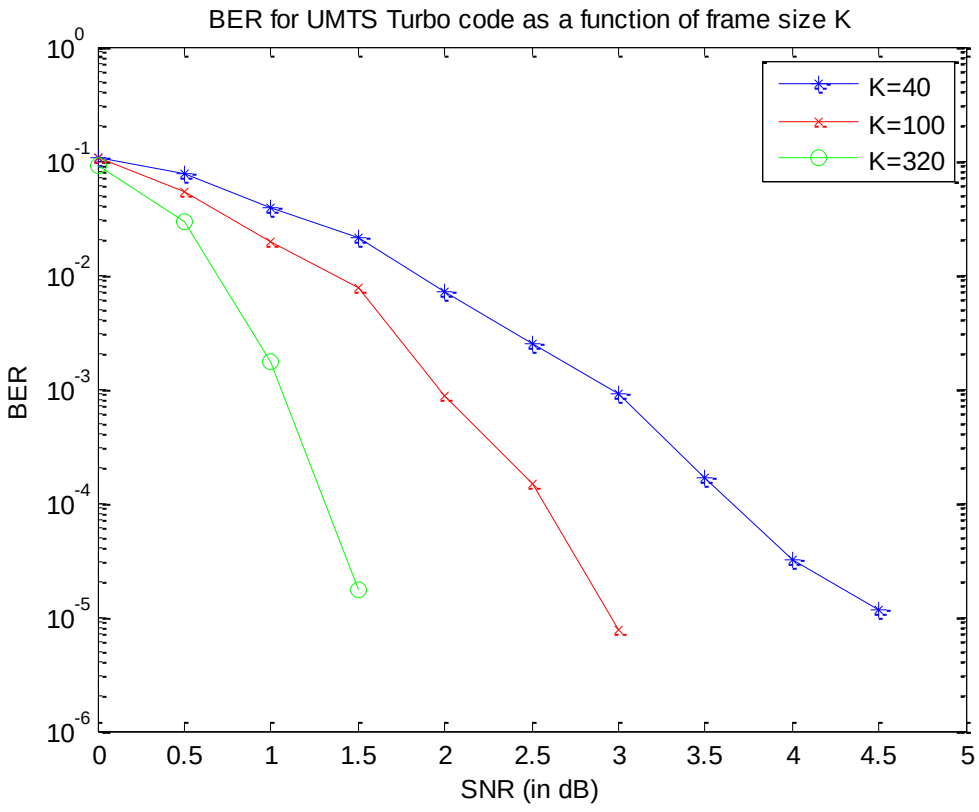


Figure 5.5 Performance comparison of UMTS turbo code after 10 decoder iterations.

5.2.4 BER performance over AWGN channel for max* algorithm

The turbo code program was simulated for frame size K = 40 over an AWGN channel. The number of frames for each SNR was taken as 500 and the code was executed for 10 decoder iterations. Decoding was done using all four variations of the max* algorithm. i.e., log MAP, max-log MAP, constant-log MAP, linear-log MAP. For performance comparison, the BER after 10 iterations is plotted in Figure 5.6. The BER values after 10 decoder iterations are given in Table 5.4.

Table 5.4 BER for frame size K = 40 after 10 decoder iterations.

Algorithm	BER
Log MAP	1.1519×10^{-5}
Max-log MAP	1.8967×10^{-5}
Constant-log MAP	0.7137×10^{-5}
Linear-log MAP	1.0358×10^{-5}

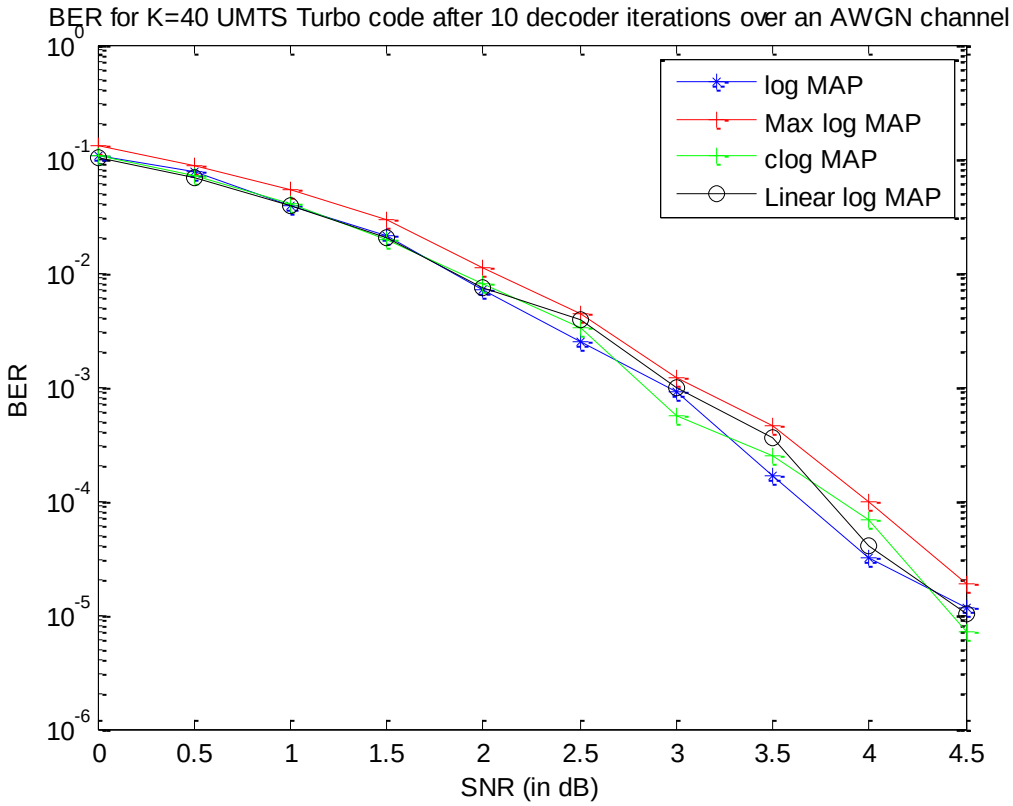


Figure 5.6 BER for frame size K=40 UMTS turbo code after 10 decoder iterations over an AWGN channel.

The figure shows that the code was able to achieve BER as low as 10^{-5} for the input bits. The log MAP, linear log MAP and constant log MAP algorithms show indistinguishable performance for the specified frame size K=40. However, in practice the performance of the log MAP algorithm is the best of all. This also hold true in our simulation when the frame size K is increased. However, the code has been simulated for K=40 as larger frame sizes take much longer time for the simulations to run.

It can be seen that performance of the max-log MAP algorithm is significantly worst of the four algorithms. However, it has the least complexity and it takes the smallest time to simulate. Thus, there is a trade off between complexity and performance.

In the figure, it is observed that the performance of linear-log MAP and constant-log MAP is sometimes slightly better than log MAP. This is an interesting and unexpected phenomenon. The reason for this is that the two constituent decoders are optimal in terms of minimizing the local BER but the overall turbo decoder does not guarantee minimizing the global BER [26]. This is because of the random perturbations due to approximation in computing the partial path metrics and LLRs in constant log MAP and linear-log MAP algorithm. These perturbations are minor and the results of constant-log MAP and linear-log MAP are very close to the log MAP algorithm.

5.2.5 BER performance over Rayleigh channel for max* algorithm

The turbo code program was simulated for frame size K=40 over a Rayleigh fading channel. The specifications were kept similar to the previous simulation. The BER after 10 iterations is plotted as a function of SNR in Figure 5.7.

After 10 decoder iterations, the log MAP algorithm achieved BER of approximately 0.74×10^{-3} at SNR 4.5 dB. In case of AWGN channel, the BER achieved was approximately 10^{-5}. As seen earlier, the performance of the turbo code over Rayleigh channel for a specified SNR is worse as compared to AWGN channel. Also, the BER for max-log MAP is the highest of all and that of log MAP is the lowest of all. Thus, the log MAP algorithm shows the best performance at the cost of highest complexity and max-log MAP shows the worst performance but it is also the least complex of all. The complexity and performance of constant-log MAP and linear-log MAP are between the log MAP and max-log MAP algorithms.

Table 5.5 BER for K = 40 over a Rayleigh fading channel after 10 decoder iterations.

Algorithm	Log MAP	Max-log MAP	C-log MAP	Linear-log MAP
BER	0.74×10^{-3}	0.14×10^{-2}	0.82×10^{-3}	0.84×10^{-3}

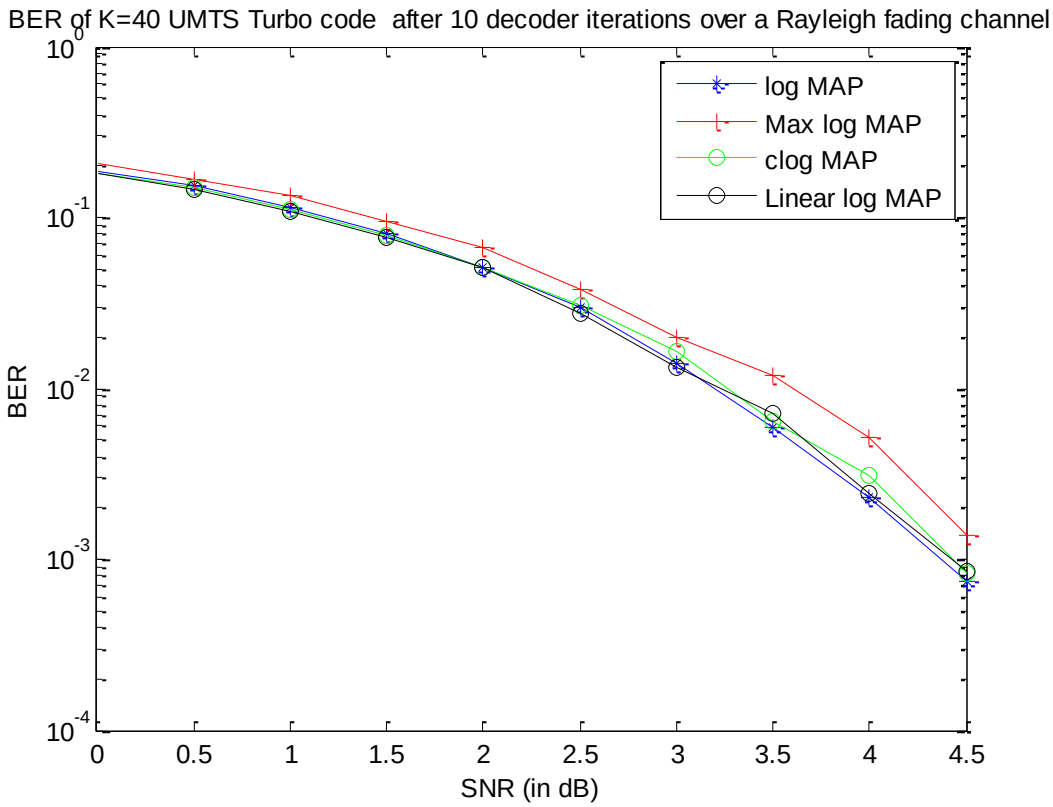


Figure 5.7 BER for frame size K=40 UMTS turbo code over a Rayleigh fading channel after 10 decoder iterations.

5.2.6 Performance comparison over Rayleigh and AWGN channel

For comparison, the BER curves for AWGN and Rayleigh fading channels are plotted together. Figure 5.8 shows the comparison. The plots are shown for SNR range 0 to 4.5 dB. The code is able to achieve much lower BER at a specified SNR over an AWGN channel as compared to Rayleigh fading channel. The BER of approximately 10^{-2} was achieved over Rayleigh fading channel at SNR 3 dB. For an AWGN channel, the same BER was achieved at SNR 1.75 dB. Thus, higher SNR or more number of iterations is required to achieve the same performance of AWGN channel over Rayleigh fading channel.

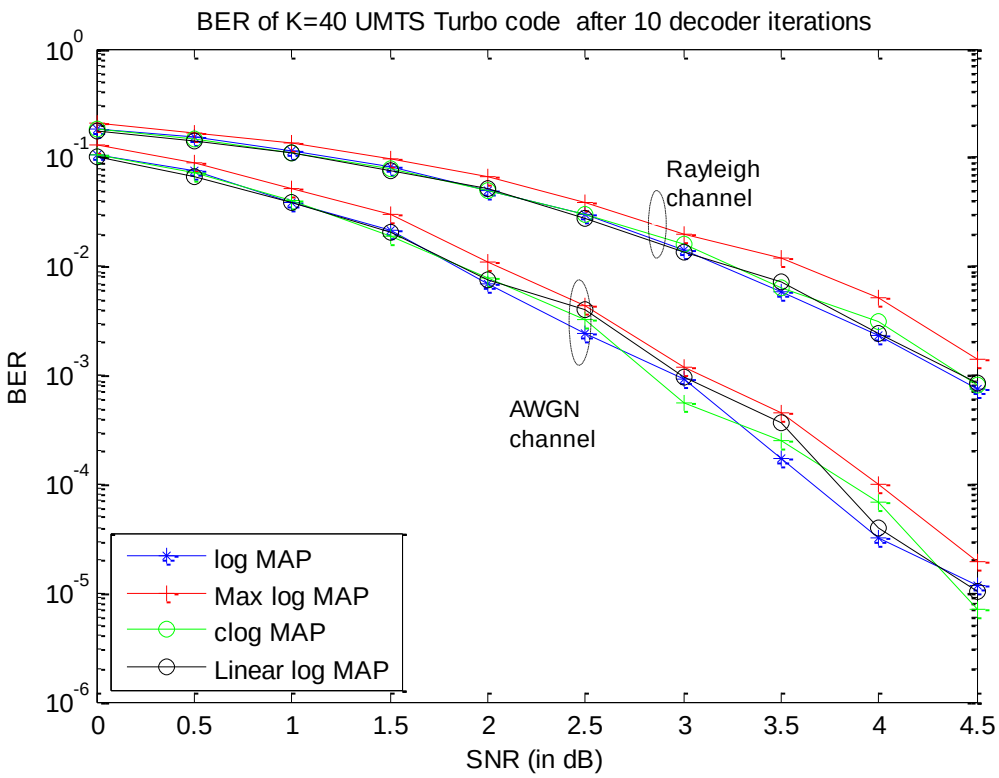


Figure 5.8 BER for frame size K=40 UMTS turbo code after 10 decoder iterations.

5.2.7 BER as a function of number of iterations

The turbo code performance improves with the increase in the number of iterations. However, after a specific number of iterations, the BER stays constant and does not decrease any further. For achieving lower BER either the SNR or frame size K needs to be changed. By increasing the SNR, the signal power increases. As a result the *a priori* information of the data available improves and it results in better decoding of the data. Thus, lower BER can be achieved. Similarly, when frame size K increases, the interleaver adds more randomness to the data. This also results in better decoding and lower BER.

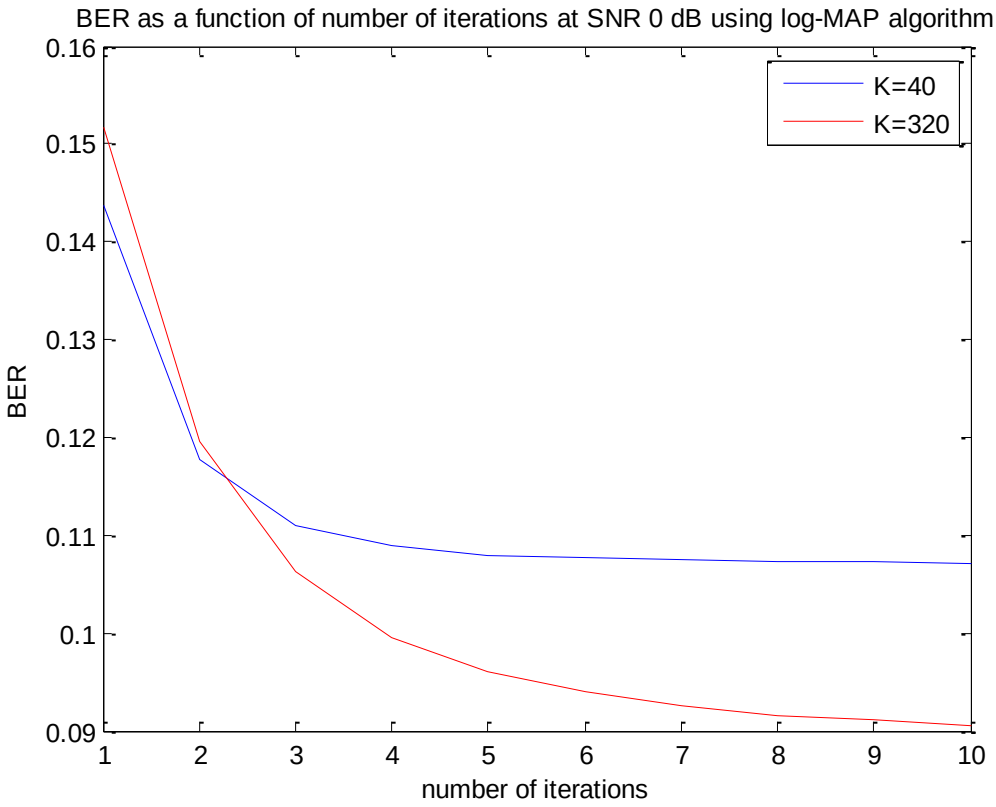


Figure 5.9 The evolution of BER as a function of number of iterations.

Figure 5.9 shows the evolution of BER as a function of number of iterations at SNR 0 dB. It can be seen that for frame size K 40, the BER is much higher as compared to the frame size K 320. These plots show that the decoder is well implemented as the BER is evolving after every iteration. Also the BER decreases exponentially and approach a very low value as the number of iteration increases.

5.3 EXIT chart analysis

The BER chart for iterative decoding has three parts;

- The region with negligible BER reduction.
- The turbo cliff with persistent BER reduction over many iterations.
- The BER floor where low BER can be reached after just few iterations.

Although bounding techniques are used to terminate the code and avoid extra computations, still BER analysis is not enough for performance analysis of the code. This is because the bounding techniques are not perfect and they have some limitations [39].

EXIT chart is used to visualize the flow of information transfer through the constituent decoders. It is useful for performance analysis of turbo decoder in low BER regions where the BER curve was staying constant. For doing so, the mutual information is plotted between the extrinsic information at the output of the constituent decoder and the input message with respect to the mutual information between the *a priori* information at the input of the constituent decoder and the message.

To account for the iterative nature of the suboptimal decoding algorithm, both decoder characteristics are plotted into a single figure. For doing so, the transfer characteristics of the second decoder the axes are swapped. This diagram is referred to as EXIT chart. The exchange of extrinsic information can be visualized as a decoding trajectory between the two curves. The exit chart for frame size K 320 at SNR 0 dB is shown in Figure 5.10. The simulation was run for 100 frames and then the mean extrinsic information was plotted. The dotted lines show the standard deviation of the extrinsic information about the mean.

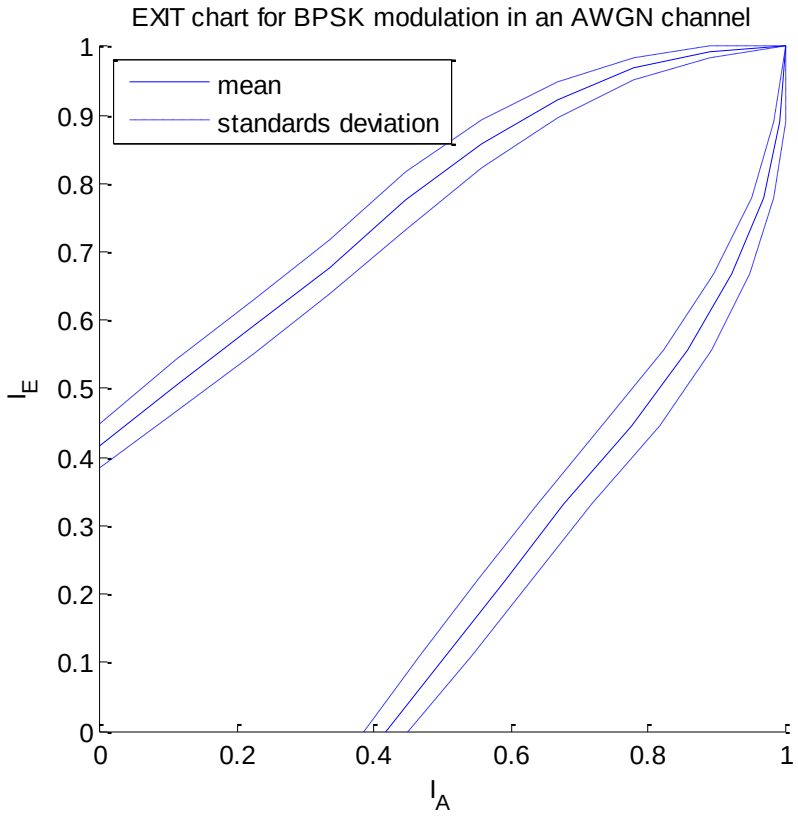


Figure 5.10 EXIT chart showing mean and standard deviation of turbo code.

5.3.1 Extrinsic information as a function of *a priori* information

Extrinsic information was plotted expressed as a function of *a priori* information for different SNR values in Figure 5.11. It can be seen that as the SNR increases, the extrinsic information IE increases. As a result, we obtain higher values of extrinsic information IE even when IA is null. When the channel is good .i.e., SNR is high; the data is less corrupted after passing through the channel. As the input to the decoder now contains lesser errors, so even if the *a priori* information is 0, the decoder can still output a message that makes sense. Thus, the decoder is able to perform better with lower BER. On the contrary, when the channel is noisier, the decoder fails to estimate the message without efficient *a priori* information.

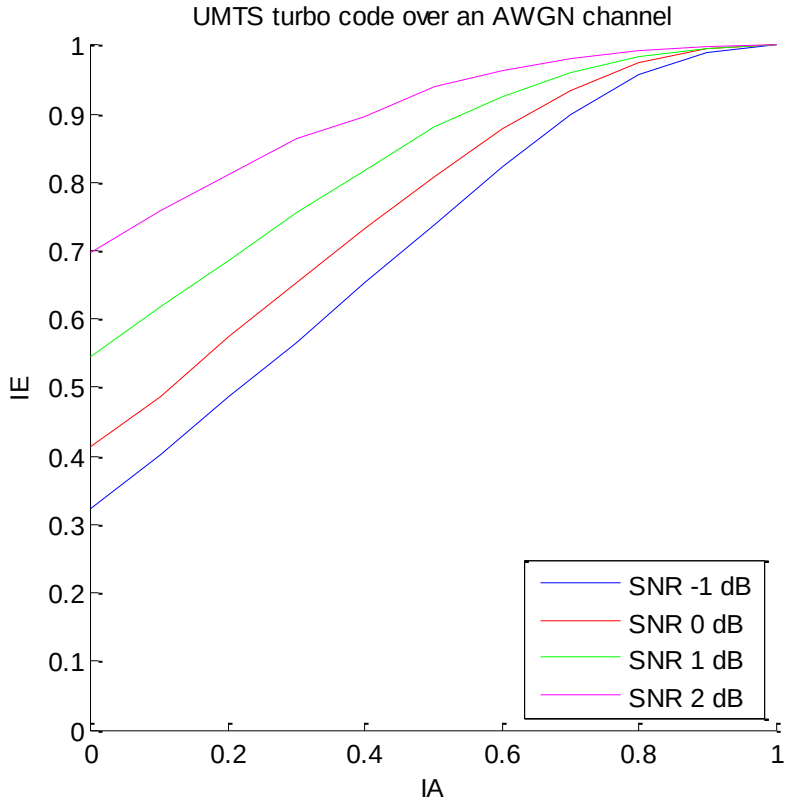


Figure 5.11 Extrinsic information expressed as a function of a priori information.

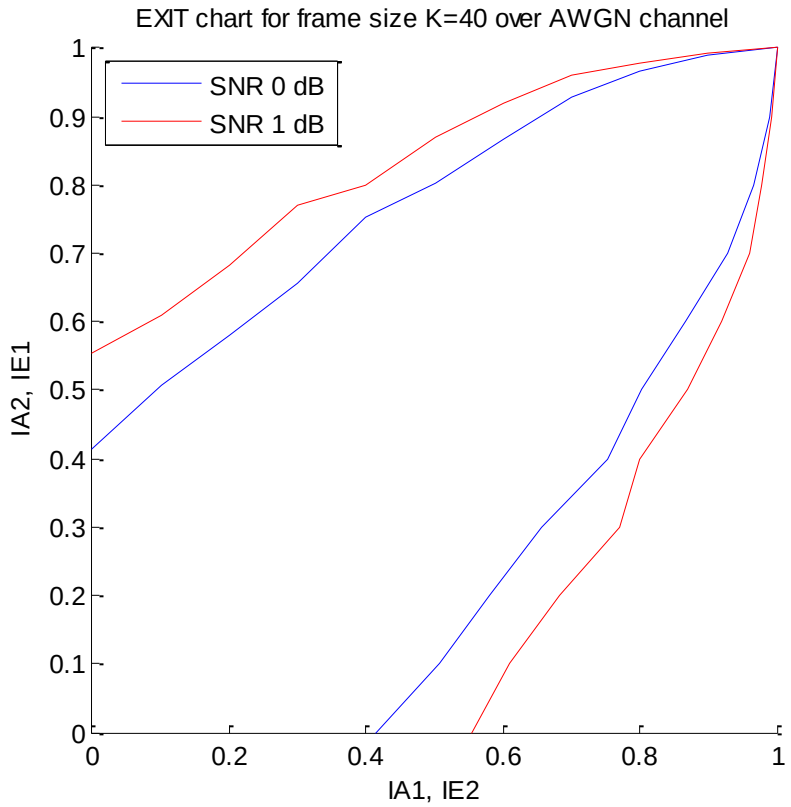


Figure 5.12 The Extrinsic information plot as a function of SNR.

The EXIT chart for SNR 0 and 1 dB is shown in Figure 5.12. The figure shows that for SNR 1 dB the extrinsic information is higher as compared to *a priori* information. Thus, as discussed earlier, decoder is better able to decode the data.

5.3.2 Decoder trajectory

The decoder trajectory gives the visualization of the exchange of extrinsic information in the EXIT chart. The trajectory of the decoder for frame size K 40 turbo code is shown in Figure 5.13. It can be seen that with 4 iterations, the trajectory of extrinsic information and *a priori* information reaches the point approx (0.7, 0.65) on the graph. As the number of iterations is increased from 4 to 10, the trajectory reaches the point approx (0.85, 0.85). Increasing the decoder iteration further to 20, there is not much improvement in the *a priori* and extrinsic information. An increase of 10 decoder iterations result only in a very small improvement in the extrinsic information and the trajectory is not moving any further to (1, 1) point. It is rather oscillating around the same point. Thus, the iterations need to be chosen keeping in mind the trade-off between complexity and system performance.

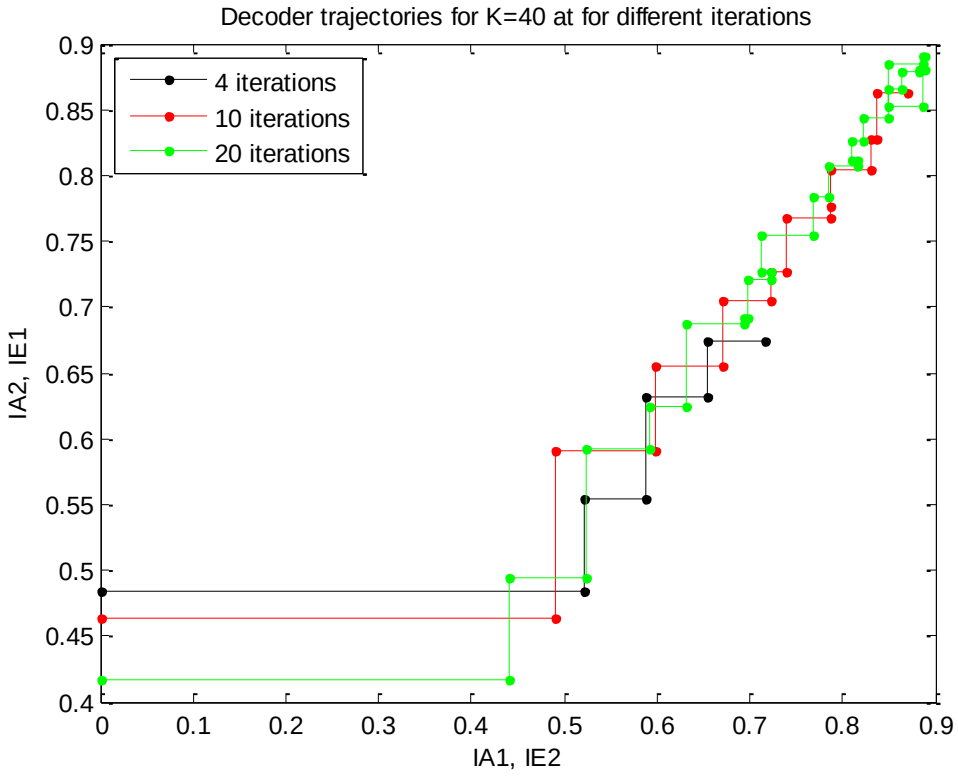


Figure 5.13 Decoder trajectory for frame size K=40 for different number of iterations.

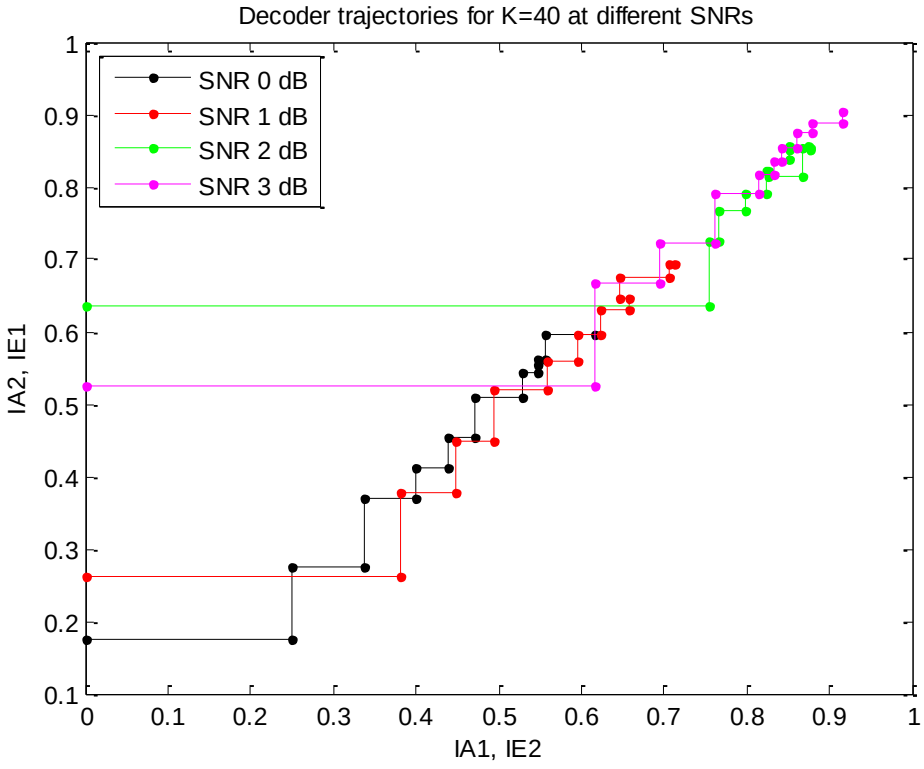


Figure 5.14 Decoder trajectory for frame size K=40 turbo code after 10 iterations.

The decoder trajectories for frame size K 40 after 10 decoder iterations for different SNRs is shown in Figure 5.14. As the SNR increases, the *a priori* information to decoder 1 increases. This also results in increased extrinsic information. There is an improvement in system performance shown by the improvement in extrinsic information 2 at the end of the iterations. As the SNR increases from 0 to 2 dB, the extrinsic information of decoder 2 at the end of the 10^{th} iteration improves significantly from 0.65 to 0.9. On further increase in SNR from 2 dB to 3 dB, the extrinsic information shows very little improvement. The decoder trajectory for frame size K 320 after 20 decoder iteration shows similar results as shown in Figure 5.15.

It can be seen that with the increase in SNR from 0 to 2 dB, the extrinsic information after 20 iterations improves from 0.65 to 0.85 approx. However, the later iterations don't show much improvement in the performance. Thus, there is a trade-off between energy efficiency, complexity and performance of the system.

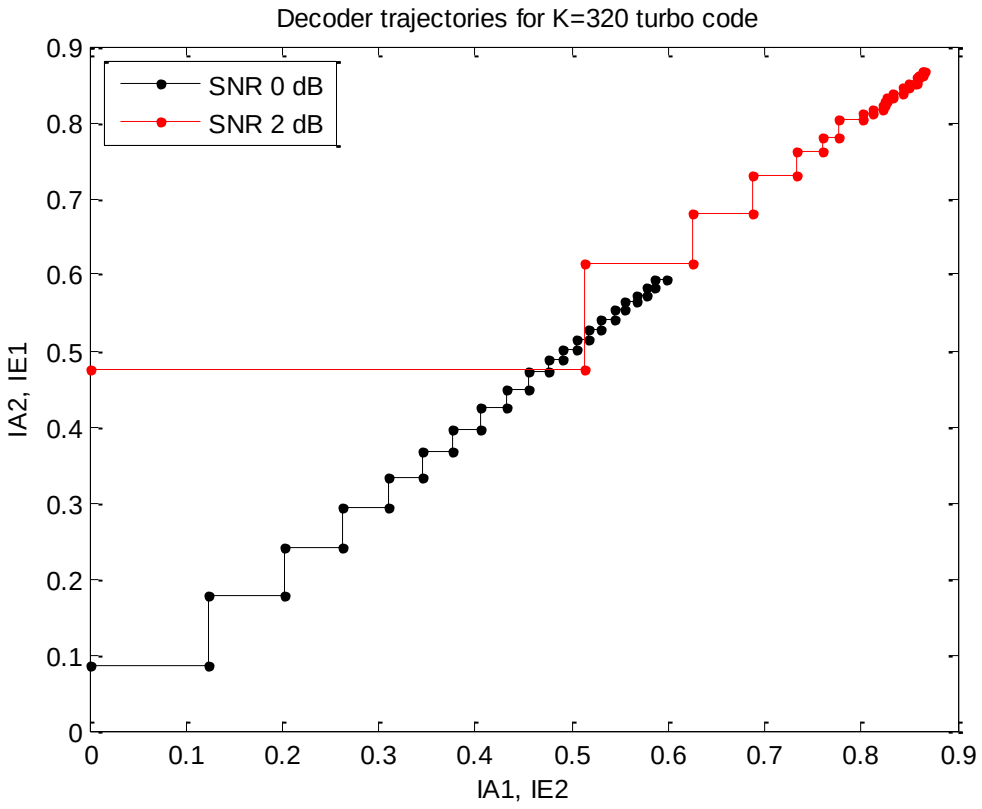


Figure 5.15 Decoder trajectory for K=320 turbo code after 20 decoder iterations.

5.4 Comparison with simulation results from MATLAB coded modulation library (CML)

CML is an open source toolbox for the simulation of capacity approaching codes in MATLAB. It is available at the iterative solutions website [40]. It mainly runs in MATLAB but the computationally intensive parts are simulated in C. It thus uses c-mex files for simulations. A large database of previously simulated results is also available at the website in the form of MAT files. These results are the result of many hours of simulation run time. New simulations can be done using the *CmlSimulate* command in MATLAB. A number of scenarios are given in the *UmtsScenarios.m* specifying the frame size, number of iterations, channel model, etc. For better performance comparison, the CML library code and the MATLAB implementation done in the thesis were run for similar input parameters. The results of the CML library are stored in MAT file and are plotted using the *CmlPlot* command along with the scenario number.

5.4.1 Turbo code performance comparison for frame size K = 40 using log MAP algorithm

The turbo code is simulated for frame size K 40 using log MAP algorithm. The number of iterations was set to 10. The SNR range was specified from 0 to 4.5 dB. The maximum trials and the minimum BER was kept 10^6 and 10^{-6} respectively for the CML simulation. The turbo code implementation was also run for similar conditions. The BER curves for 10 decoder iterations for CML simulation are shown in Figure 5.16. The results from the implemented code are shown in Figure 5.17.

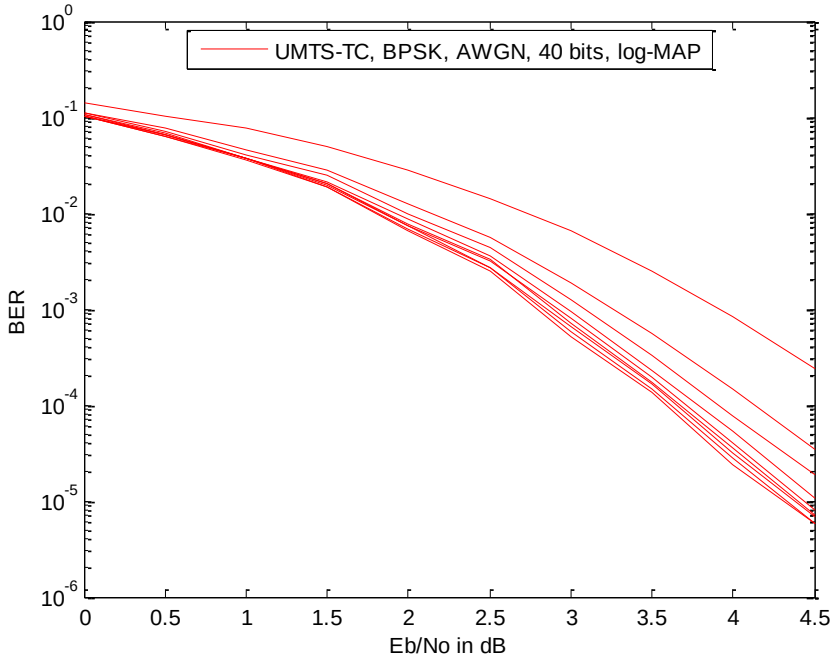


Figure 5.16 BER curves for frame size K 40 using log MAP obtained from CML simulation.

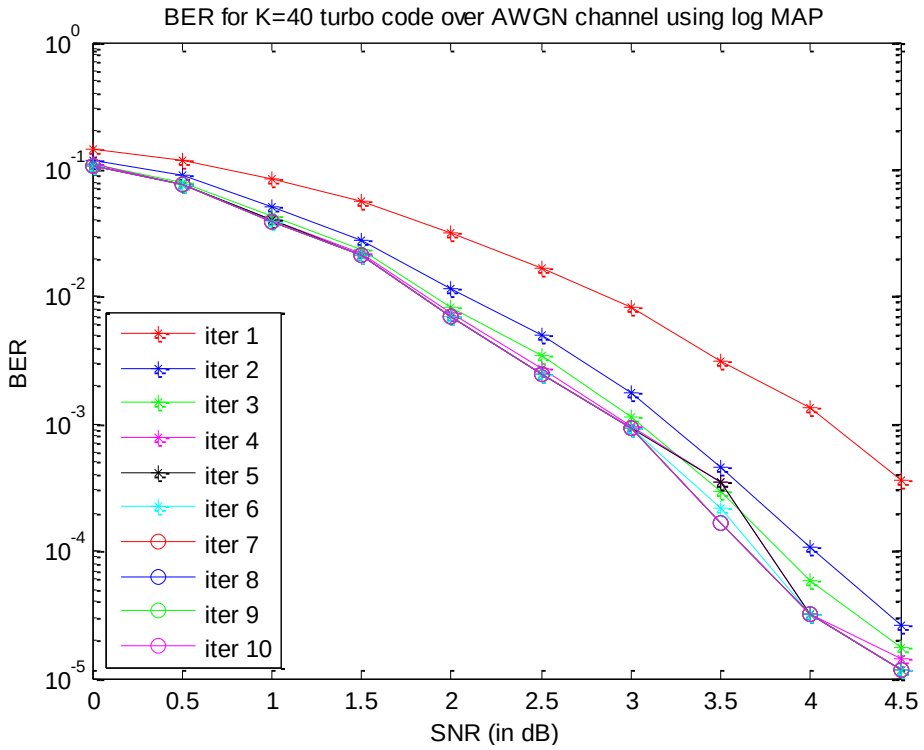


Figure 5.17 BER curves for frame size K 40 using log MAP obtained from simulated Turbo code.

The CML simulation achieved a BER between 10^{-3} and 10^{-4} at 4.5 dB after first decoder iteration. The BER decreases with the increase in number of iterations. At the fourth iteration, the BER reaches 10^{-5} but after that the BER change is very small for the next iterations. After the tenth iteration, the BER is close to 10^{-5}. The turbo code implementation done in the thesis also shows similar results. Figure 5.17 shows that the BER for turbo code after first iteration is 5 x 10^{-3} at SNR 4.5 dB. The BER falls with the increase in number of iterations and at the end of tenth iteration the BER is 1.34 x 10^{-5}. This is very close to the BER achieved by the CML simulation. Thus, the implemented turbo code performance is very close to the CML simulation.

5.4.2 Turbo code performance comparison for frame size K = 40 using max-log MAP algorithm

The turbo code simulation was run for frame size K = 40 over an AWGN channel using 10 decoder iterations. The decoding algorithm used was max-log MAP. The simulation results for the CML and the simulated code are shown in Figure 5.18 BER for frame size K = 40 using max-log MAP obtained from CML simulation.Figure 5.18 and Figure 5.19 respectively. The CML simulation achieves a BER of 10^{-5} at SNR 4.5 dB. The turbo code simulation was able to achieve a BER of 0.1896 x 10^{-4}. Once again, the results of the simulation are very close to the CML simulated results.

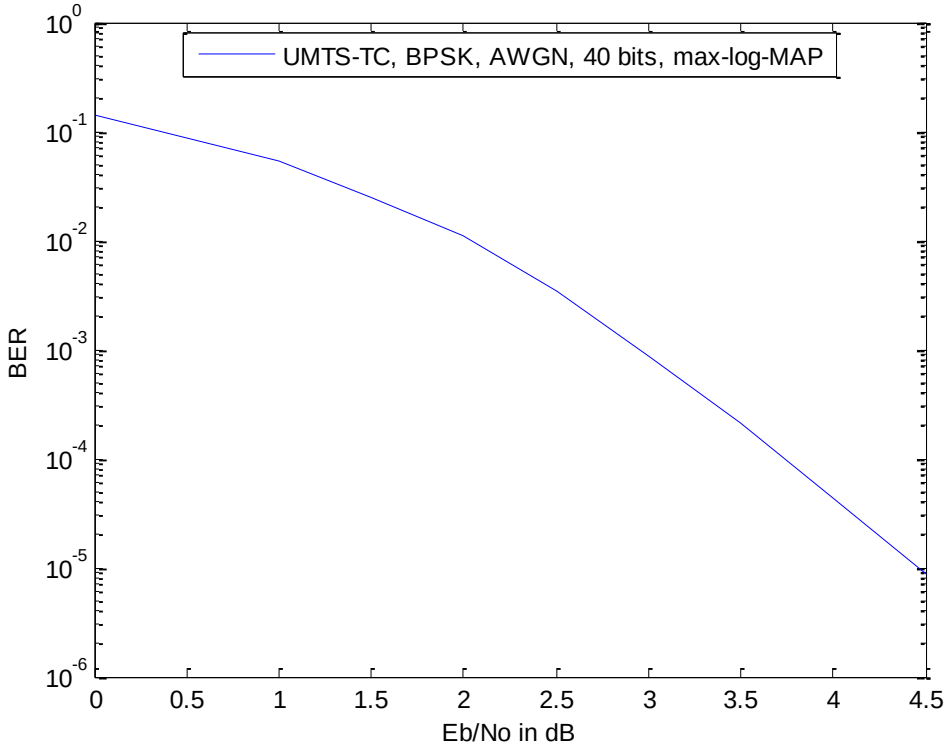


Figure 5.18 BER for frame size K = 40 using max-log MAP obtained from CML simulation.

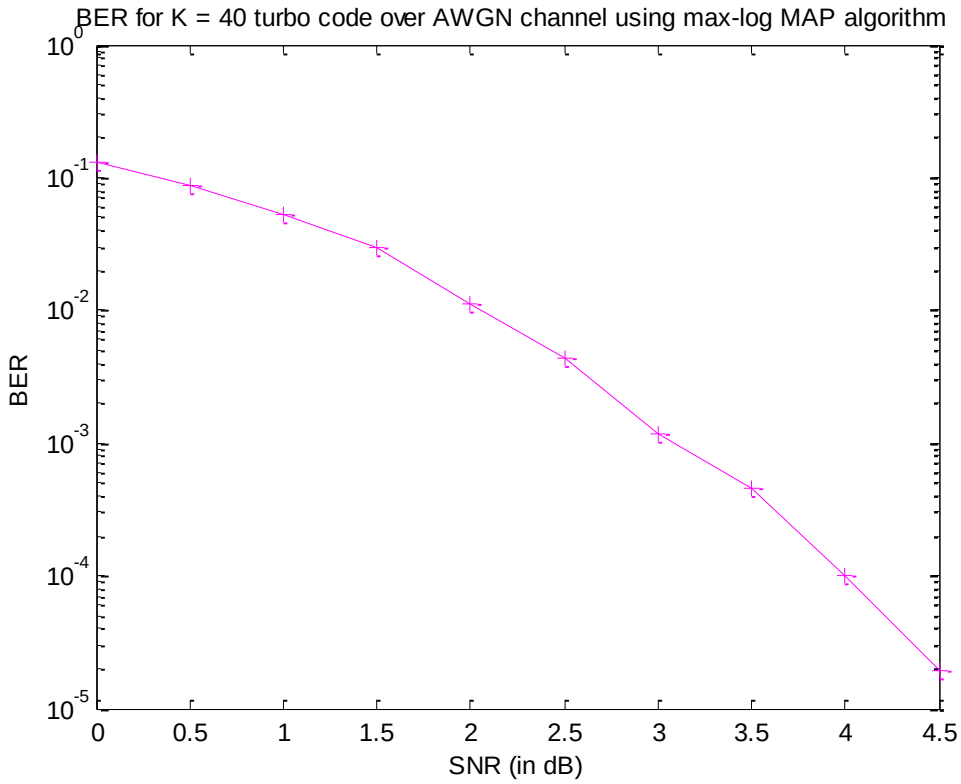


Figure 5.19 BER for frame size K = 40 using max-log MAP obtained from simulated Turbo code.

5.4.3 Turbo code performance comparison for frame size K = 40 over Rayleigh channel

The CML simulation and MATLAB turbo code were run for frame size K = 40 over Rayleigh fading channel. The decoding algorithm was log MAP and the number of decoder iterations was chosen as 10. The results of the CML code and the turbo code simulation are shown in Figure 5.20 and Figure 5.21 respectively. The BER for both the simulations is approximately 10^{-4} at SNR 5 dB.

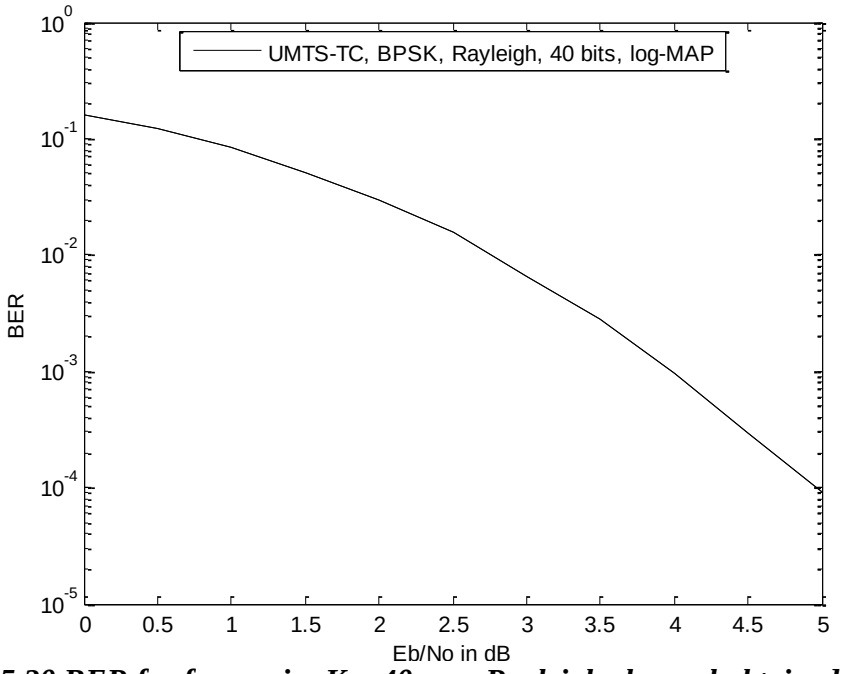


Figure 5.20 BER for frame size K = 40 over Rayleigh channel obtained from CML simulation.

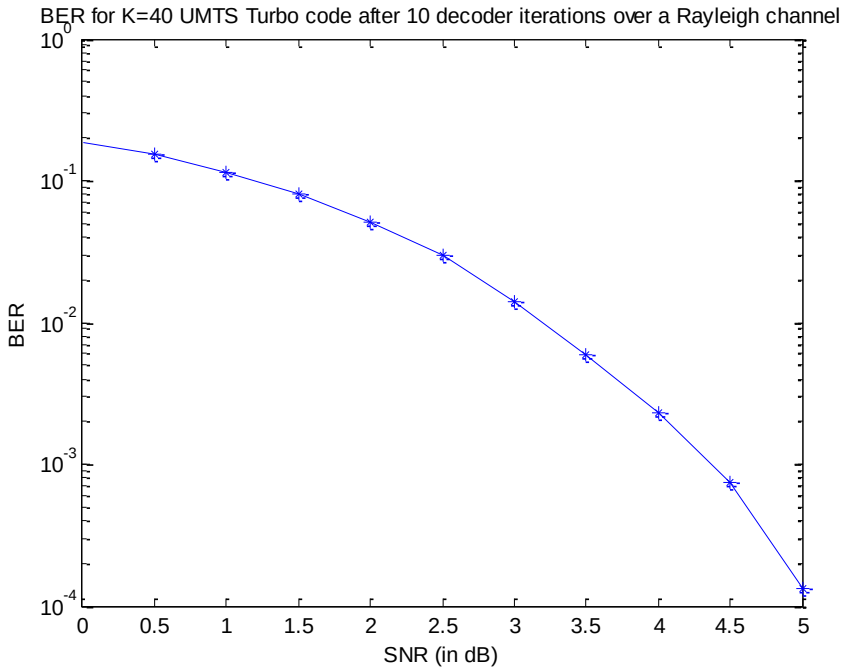


Figure 5.21 BER for frame size K = 40 over Rayleigh channel obtained from simulated turbo code.

5.4.4 Turbo code performance comparison for frame size K = 100

The turbo code was simulated for frame size K = 100 over an AWGN channel using log MAP algorithm. The number of decoder iterations was chosen as 10. The code was run for maximum trials 10^6 and minimum BER 10^{-6}. The results obtained from the CML simulation are shown in Figure 5.22. The turbo code simulation was run with similar parameters. The results are shown in Figure 5.23. The curves show similar trends and the BER at different SNRs are also close enough. Some variations are due to the fact that the input was generated randomly and was different for both the simulations.

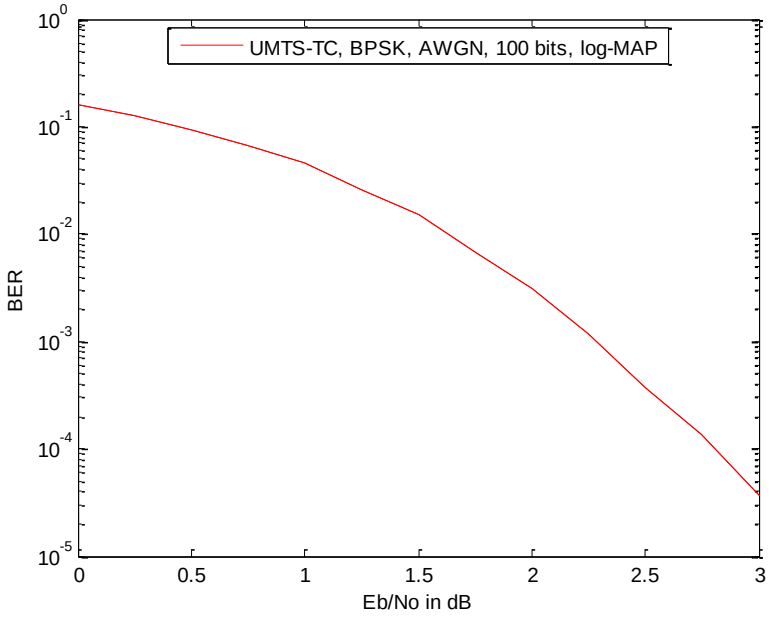


Figure 5.22 BER for frame size K = 100 using log MAP from CML simulation.

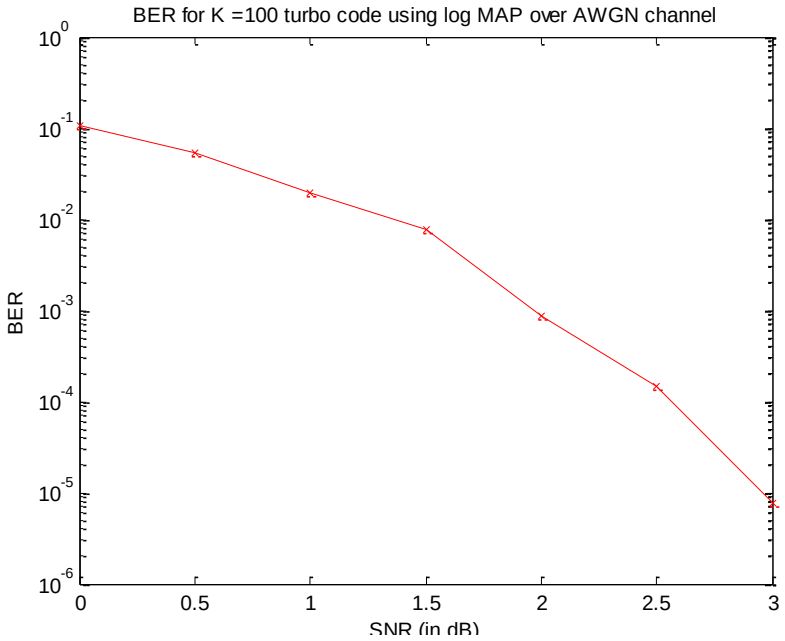


Figure 5.23 BER for frame size K = 100 using log MAP from simulated turbo code.

5.4.5 Turbo code performance comparison for frame size K = 320

The turbo code was simulated for frame size K = 320 over an AWGN channel using 10 decoder iterations. The results obtained from the CML code and the simulated turbo code are shown in Figure 5.24 and Figure 5.25 respectively. The BER for the CML simulation was between 10^{-4} and 10^{-5} at SNR of 1.5 dB. The simulated turbo code achieved a BER of 1.75 x 10^{-5}. Thus, the performance of the simulated turbo code is very close to the CML simulation.

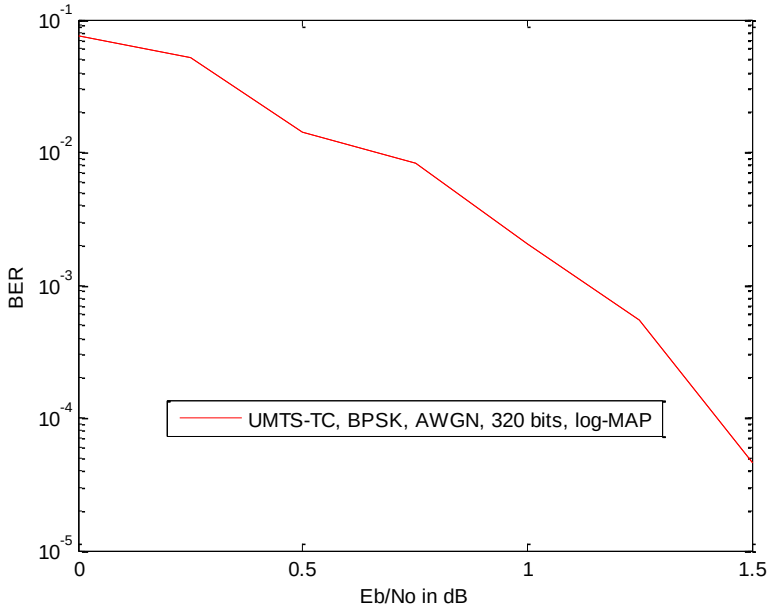


Figure 5.24 BER for frame size K = 320 over AWGN channel from CML simulation.

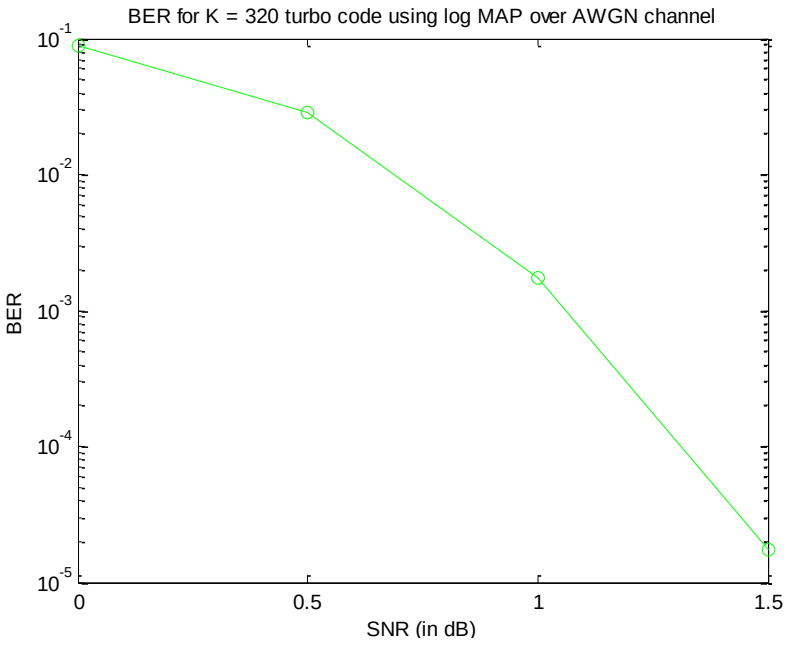


Figure 5.25 BER for frame size K = 320 using simulated Turbo code.

6. SUMMARY AND CONCLUSIONS

The thesis aims at studying and analysing the performance of turbo codes. In the thesis, turbo codes used in the UMTS standard were implemented in MATLAB. This chapter summarizes the work done and then draws the conclusion based on the obtained simulation results.

6.1 Summary

Turbo codes are a class of high performance error correction codes with error performance close to the Shannon limit. They make use of the parallel concatenated recursive coding to provide enough error protection with reasonable complexity. The basic turbo code encoder consists of two RSC encoders connected in parallel. The RSC encoders produce higher weight code words as compared to the non RSC encoders and are thus better suited for turbo codes. An interleaver is used between the two encoders to introduce randomness as random codes achieve better channel capacity. The output of the encoder consists of systematic bits as well as encoded bits from the two encoders.

The decoder takes into account the received codeword along with the knowledge of the code structure to compute the LLRs. Decoding is performed in an iterative fashion with the information exchanged among the constituent encoders for better performance. The decoder first computes the branch metrics for all the nodes of the trellis. The next step is to compute the reverse path metric for the received frame of data and the result is stored in memory. After that, the forward path metrics are calculated. At the end the LLR of the data for the whole frame is calculated and the information is passed to the other decoder. After the end of the iterations, a hard decision is taken on the LLR to get the received data bits.

The RSC decoder uses a modified Jacobi algorithm for sweeping through the trellis. Four versions of the Jacobi algorithm .i.e., log MAP, max-log MAP, c-log MAP and linear-log MAP, were considered in the implementation. Performance comparison was also done based on the BER curves. The algorithm is implemented twice in every half iteration, and thus constitutes a major portion of the decoding complexity.

6.2 Conclusions

Turbo codes are being used in 3G and 4G mobile telephony, WiMAX and satellite communication systems. They show extraordinary performance at low SNRs approaching the Shannon limit. However at higher SNRs, the BER curve begins to flatten and hinders the ability to achieve extremely small bit error rates. The simulation results showed that the performance of the turbo code depends on a number of parameters including the frame size K, number of decoder iterations, the SNR and the choice of log MAP algorithm.

Turbo codes are iterative codes and the performance improves with the increase in the number of iterations. Typically 5 to 10 iterations produce most of the improvement. In the simulated results, the BER dropped significantly from the first to the fifth iteration. However, there was not much improvement from the fifth to the tenth iteration. Thus, we can conclude that as the number of iterations increases, the rate of performance improvement decreases. For achieving lower BER either the SNR or frame size K needs to be changed.

When the signal power is increased, the SNR increases. As a result the *a priori* information of the data available improves. As the input to the decoder now contains lesser errors, the decoder can still output a message that makes sense thus, lower BER can be achieved.

The performance of the turbo code profoundly depends on the randomness introduced by the interleaver. An increased interleaver size K results in higher probability of higher weight code words, thus resulting in better decoder performance. This was shown in the simulation as the BER after 10 decoder iterations dropped from 10^{-2} to 10^{-5} by increasing the frame size K from 40 to 320. Thus, by increasing the frame size K, lower BER can be achieved by keeping the SNR constant but at the cost of increased latency. Also, the code is able to achieve much lower BER at a specified SNR over an AWGN channel as compared to Rayleigh fading channel.

The four variations of the classical log MAP algorithm were implemented. It was seen that the performance of the max-log MAP algorithm was significantly worst of the four algorithms and the log MAP algorithm had the best performance. However, the max-log MAP has the least complexity and it takes the smallest time to simulate. Thus, there is a trade off between complexity and performance. Although the log MAP algorithm is more complex but it can achieve similar BER performance as that of the max-log MAP algorithm using less iteration.

Thus, we can conclude that in designing turbo codes there is a trade-off between energy efficiency, bandwidth efficiency, latency, complexity and error performance.

REFERENCES

[1] Muhammad Imran Anwar and Seppo Virtanen and Jouni Isoaho: "a software defined approach for common baseband processing", *Journal of Systems Architecture*, Jan. 2008.

[2] S. Weiss, A. Shligersky, S. Abendroth, J. Reeve, L. Moreau, T. E. Dodgson and D. Babb: "A software defined radio testbed implementation", *IEE Colloquium on DSP Enable Radio*, Livingston, Scotland, 2003, pp. 268-274.

[3] Walter H.W. Tuttlebee, "Software defined radio: Facets of a developing technology", *IEEE Personal Communication*, 6(2):38-44, April 1999.

[4] Z. Salcic, C. F. Mecklenbrauker, "Software radio - architectural requirements, research and development challenges", *8th International Conference on Communication Systems*, Vol 2, Nov. 2002.

[5] Petri Isomäki, Nastooh Avessta, "An Overview of Software Defined Radio Technologies", TUCS Technical Report No 652, Turku, Finland, Dec. 2004.

[6] Bernard Sklar, Digital Communications: Fundamentals and Applications, Second Edition, Prentice Hall (2001).

[7] Fu-hua Huang, "Evaluation of Soft Output Decoding for Turbo Codes", Virginia Polytechnic Institute and State University, Master's thesis, 1997.

[8] Core technologies, Viterbi algorithm for decoding of convolutional codes, http://www.1-core.com/library/comm/viterbi/, cited May, 2012.

[9] Emilia Käsper, "Turbo Codes", www.tkk.fi/~pat/coding/essays/turbo.pdf, cited 15 June, 2012.

[10] C. Berrou, A. Glavieux, and P. Thitimajshima., "Near Shannon limit error correcting coding and decoding: Turbo codes", *IEEE International Conference on Communications*, Geneva, Switzerland, May 2003.

[11] G. Battail, C. Berrou, and A. Glavieux, "Psuedo-Random recursive convolutional coding for near-capacity performance", *Globecom 1993*, pp. 23-27, Dec. 1993.

[12] Multu Koca, "Turbo space time equalization of broadband wireless channels", University of California, Davis, 2001.

[13] R.V. Rajakumar, Saswat Chakrabarti, Channel coding, http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Digi%20Comm/pdf-m-6/m6l35.pdf, cited 22 Feb. 2012.

[14] Shu Lin and Daniel J. Costello, Error Control Coding: Fundamentals and Applications, Prentice Hall, 1983.

[15] D. Divsalar and F. Pollara, "Turbo codes for deep-space communications", *JPL TDA Progress Report 42-120*, Feb. 15, 1995

[16] D. Divsalar and F. Pollara, "Turbo codes for PCS applications", *Proceedings of ICC 1995*, Seattle, Washington, pp. 54-59, June 1995.

[17] S. Benedetto and G. Montorsi, "Unveiling turbo codes: some results on parallel concatenated coding schemes," *IEEE Transactions on Information Theory*, Vol. 42, No. 2, pp. 409-428, March 1996.

[18] J. G. Proakis, "Digital Communications", 3rd ed., New York, McGraw-Hill, 1995.

[19] P. Jung and M. Nasshan, "Performance evaluation of turbo codes for short frame transmission systems", *Electronics Letters*, Vol. 30, No. 2, pp. 111-113, Jan. 20, 1994.
[20] Shobha Rekh, S. Subha Rani, A. Shanmugam, "Optimal choice of interleaver for turbo codes", *Academic Open Internet Journal*, volume 15, 2005.
[21] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and non-random permutations," *JPL TDA Progress Report 42- 122*, Aug. 15, 1995.
[22] A. S. Barbulescu and S. S. Pietrobon, "Interleaver design for turbo codes", *Electronics Letters*, Vol. 30, No. 25, pp. 2107-2108, Dec. 8, 1994.
[23] Charan Langton, "Turbo coding and MAP decoding", intuitive guide to principles of communications.
http://www.complextoreal.com/chapters/turbo1.pdf
[24] Bernard Sklar, "Maximum aposteriori Decoding of turbo codes", online paper.
[25] European Telecommunications Standards Institute, "Universal mobile telecommunications system (UMTS): multiplexing and channel coding (FDD), 3GPP TS 125.212 version 8.6.0, Release 8, pp. 18-23, Sept. 2009.
[26] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software-defined radios", *International Journal of Wireless Information Networks*, Vol. 8, no. 4, pp. 203–215, Oct. 2001.
[27] Teodor Iliev, "A study of turbo codes for UMTS third generation cellular standard", International Conference on Computer Systems and Technologies - *CompSysTech'07,* VI. 6.
[28] Matthew C. Valenti and Jian Sun, Handbook of RF and wireless technologies, "Turbo codes", Chapter 12, 375–399. F. Dowla Editor, Newnes Press, 2004.
[29] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate", *IEEE Trans. Inform. Theory*, Vol. 20, pp. 284–287, Mar. 1974.
[30] P. Robertson, P. Hoeher, and E. Villebrun, "Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding", *European Transactions on Telecommunications (ETT),* Vol. 8, pp. 119–125, 1997.
[31] A. J.Viterbi, "An intuitive justification and simplified implementation of the MAP decoder for convolutional codes", *IEEE Journal on Selected Areas of Communication,* Feb. 1998.
[32] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm", *IEEE Transactions on Information Theory,* Vol. 13, pp. 260–269, Apr. 1967.
[33] W. J. Gross and P. G. Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoders", *Electronics Letters,* Vol. 34, pp. 1577–1578, Aug. 6, 1998.
[34] Matthew C. Valenti, "An efficient software radio implementation of the UMTS turbo codec", *IEEE 12th International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC '01),* vol. 2, pp. G108-G113, San Diego, USA, Sep. 2001.
[35] J. F. Cheng and T. Ottosson, "Linearly approximated log-MAP algorithms for turbo coding", *IEEE Vehicular Technology Conference. (VTC)*, Houston, TX, May 2000.
[36] Liang Li, "Analysis of low power implementational issues of turbo-like codes in body area networks", Progress report for continuation towards PhD, University of Southampton, Nov. 2009.

[37] F. Gilbert, F. Kienle, N. Wehn, "Low complexity stopping criteria for UMTS turbo-decoders", *IEEE Vehicular Technology Conference (VTC)*, April 2003.

[38] Lajos Hanzo, Robert G. Maunder, Jin Wang and Lie-Liang Yang, "Near-Capacity Variable-Length Coding: Regular and EXIT chart aided irregular designs", Wiley, 2010.

[39] Stephan ten Brink, "Convergence behavior of iteratively decoded parallel concatenated codes", *IEEE Transactions on Communications*, vol. 49, Oct. 2001.

[40] Coded modulation library by iterative solutions, http://www.iterativesolutions.com, cited 26 Feb, 2013.

Appendix: MATLAB code

- **Main program for UMTS turbo code**

```
% The main Program for generating the UMTS Turbo code in MATLAB
clear all
close all
clc

K=100;                              % Specify frame size between 40 and
5114 bits.
iter_cnt = 10;                      % Specify the maximum number of
decoding iterations to perform.
symbols_per_frame=K*3+12;           % Symbols in frame after encoding.
rate=K/symbols_per_frame;           % Code rate of the Turbo code (ratio
of number of input bits to output bits)
frame_count_max=500;               % Maximum number of frames to be
passed to the code
bit_count_max=frame_count_max*K;                % Maximum bits at a
given SNR
error_max=10;                       % maximum number of errors (to be
used in condition later)
count_SNR=1;                        % counter for number of SNR values
results=[];                         % The ouput results matrix defined

channel_type=1;                     % channel_type can be 1 or 2(for AWGN
and Rayleigh fading respectively)
sterric=1                      % sterric can be 1 to 4 (for LogMAP, max-
logMAP, constant-logMAP, linear-logMAP)

max_fail_count = 3;                         % Specift the number of
iterations that should fail to improve the decoding before the
iterations are stopped early
err_out=[];                         % error count matrix

% specifying the SNR starting value, step size and the range.
SNR_start=0;
SNR_delta=0.5;
SNR_stop=4;

BER=1;
data=randint(1,K);    % generate random data
results=[zeros(1,iter_cnt+1) 1];
for SNR=SNR_start:SNR_delta:SNR_stop   % for loop for SNR

    error_counts=zeros(1,iter_cnt);     % initialize error count
matrix
    bit_count=0;

    % Keep running the code until enough errors observed or enough
bits encoded.
    while bit_count < bit_count_max || error_counts(iter_cnt) <
error_max

        % function call for turbo code
```

```
        [turbo_encoded]=turbo_enc(data);

        %function call to bpsk modulate the turbo coded signal
        [mod_signal]=bpsk_mod(turbo_encoded);

        % Function call to pass through the channel
        % channel type needs to be passed as function arguments
        [symbol_likelihood]= channel ( mod_signal , SNR
,channel_type,symbols_per_frame,rate);
        w_xk=zeros(1,K+3);    % feedback from second component encoder
to first component encoder. It is initialized to zero before first
iteration.

        avg_IA = 0;         % initializing the mutual information
averaging value to zero
        iteration_index = 1; % iteration index is initialized to 1
        fail_count = 0;          % number of times the code has failed

        % Run the simulation while the number of failed iterations
        % is less than max_fail_count and iteration index is less than
        % maximum ietartions
        while fail_count < max_fail_count && iteration_index <=
iter_cnt
            % function call to turbo decode the data

[tilda2_xk,xk_hat,errors,w_xk]=turbo_dec(symbol_likelihood,1,data,w_xk
,sterric);
            % The turbo code gives the error after each iteration.
            % if all the errors have been corrected in the
            % iteration, no need to carry on
            if errors == 0
                error_iter = 0;
                fail_count = max_fail_count; % set fail count to max
to terminate the iterations
            else
                % Calculate the mutual information average to check
the performance of code in the iteration.
                mutual_info = avg_mutual_inf(tilda2_xk);

                % If IA is improved assign best_errors as errors
                % otherwise increment the number of chances
                if mutual_info > avg_IA
                    avg_IA = mutual_info; % if improved, assign new
value
                    error_iter = errors;
                else
                    fail_count = fail_count + 1; % increment the
number of failures
                end
            end         % end of if-else loop

            % Accumulate the number of errors and bits that have been
simulated so far.
            error_counts(iteration_index) =
error_counts(iteration_index) + error_iter;

            % Increment the iteration_index counter for next iteration
            iteration_index = iteration_index + 1;
        end       % end of while loop for iterations
```

```
        % If the code is terminated early, assign the value of
        % error of last iteration before termination to the next
        % all iterations
        % if error(iter)=0,  assign zero to err(iter+1)...
        % if error(iter)=10, assign 10 to err(iter+1).....
        while iteration_index <= iter_cnt
            error_counts(iteration_index) =
error_counts(iteration_index) + error_iter;

            iteration_index = iteration_index + 1;
        end

        bit_count = bit_count + length(data);  % add the number of
bits encoded so far

        % Store the SNR and BERs in a matrix and display it.
        results(count_SNR,1) = SNR;
        results(count_SNR,2) = bit_count;
        results(count_SNR,(1:iter_cnt)+2) = error_counts
        %   BER=results(count_SNR,iter_cnt+2)./results(count_SNR,2); %
calculate the BER
    end         % end of while loop for erro and bit count

    count_SNR = count_SNR + 1;      % Increment the SNR counter
end     % end of for loop for SNR
results

% plotting the results
figure
for iteration_index = 1:iter_cnt
    semilogy(results(:,1),results(:,iteration_index+2)./results(:,2));
    hold on
end


% saving data in mat file
filename =
['AWGN_',num2str(K),'_',num2str(SNR_start),'_',num2str(SNR_stop),'.mat
'];
save(filename, 'results', '-MAT');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function for turbo encoder implementation**

```
function [output]=turbo_enc(xin)

% Function to turbo encode the data
% xin is the input data
% output is the coded data containing both data and flush bits

num=length(xin);   % size of input data

[enc_out,flush1]=conv_enc(xin,1);   % function call for convolutional
encoder

% interleaving
output=[];                    % defining output matrix
num_blocks=ceil(num/5114);    % determining number of data blocks
```

```
out_inter=[];                % interleaver output matrix
xin2=xin;
if num_blocks>1
    for cnt_block=1:num_blocks-1
        % interlevaing data block by block
        [y]=interl(xin ((((cnt_block-1) *5114) +1):cnt_block*5114));
% function call fot interleaver
        [enc_out_inter,flush]=conv_enc(y,0);
% convolutional encoding the interleaved data
        for cnt=1:length(y)
            output=[output xin2(cnt) enc_out(cnt) enc_out_inter(cnt)];
% storing encoded data in output matrix Xk,Zk ,Zk'
        end
        xin2=xin2(cnt+1:end);
        enc_out=enc_out(cnt+1:end);
    end  % end of cnt_block loop
end     % end of num_block loop

% interleaving the last block of data
input_interl = xin((( (num_blocks-1) *5114)  +1) :end);  % input to
interleaver
[y,interleaved]=interl(input_interl);                   % function
call to interleaver
[enc_out_inter,flush2]=conv_enc(y,1);               % convolution
encode the interleaved data

% storing encoded data in output matrix Xk, Zk, Zk'
for cnt=1:length(y)
    output=[output xin2(cnt) enc_out(cnt) enc_out_inter(cnt)];
end     % end of for



% appending flush bits Xk+1, Zk+1, Xk+2, Zk+2, Xk+3, Zk+3,
% X'k+1,Z'k+1,X'k+2, Z'k+2, X'k+3, Z'k+3

out=[];
m=length(xin);   % length of input data
% loop for writing flush bits from upper encoder  Xk+1, Zk+1, Xk+2,
Zk+2, Xk+3, Zk+3,
for cnt1=1:length(flush1)
    out=[out enc_out(m+cnt1) flush1(cnt1)];
end     % end of for

% loop for writing flush bits from lower encoder  X'k+1,Z'k+1,X'k+2,
Z'k+2, X'k+3, Z'k+3
for cnt1=1:length(flush2)
    out=[out enc_out_inter(m+cnt1) flush2(cnt1)];
end     % end of for

output=[output out];

end     % end of fuction

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function for constituent convolutional encoder**

```
function [enc_out,flush]=conv_enc(xin)

% constraint length 4 convolutional encoder
% Input data is xin
% output is encoded data enc_out
% Flush bits are flush

% intilaize the states to zero
data_in=[0 0 0];
s1=0;
s2=0;
s3=0;

enc_out=[];   % defining the output matrix of encoded data

for cnt=1:length(xin)
    input=xin(cnt);  % input bit to convolutional encoder

    out_back=xor(s2,s3);   % feedback data

    % new states of the convolutional encoder
     s3_new=s2;
     s2_new=s1;
     s1_new=xor(input,out_back);

     % output encoded bit
     out1=xor(xor(s1_new,s1),s3);

     % storing data in output array
     enc_out=[enc_out out1];

     % update the states
     s1=s1_new;
     s2=s2_new;
     s3=s3_new;
end     % end of for loop


% trellis termination
% switch go in down position
flush=[];   % initializing the flush matrix
for cnt=1:3     % loop for number of flush bits
     out_back=xor(s2,s3);   % feedback data

    % new states
     s3_new=s2;
     s2_new=s1;
     s1_new=xor(out_back,out_back);

     % output encoded bit

     out1=xor(xor(s1_new,s1),s3);   %output
     enc_out=[enc_out out1];    % storing encoded bit in output array
     flush=[flush out_back];    % flush bit
```

```
     % update ths states
     s1=s1_new;
     s2=s2_new;
     s3=s3_new;
end     %end of foor loop for flush bits

end     % end of function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function for interleaver main**

```
function[y,interleaved]=interl(Input_data)   % input data
%   40 <=k<=5114
% function for interleaving the data
% Input is Input_data
% Output is interleaved data y and interleaving matrix interleaved

 K =length(Input_data);  % length of input data
 [interleaved]=interleave(K);    % function call for generating
interleaved matrix array of specific size accoring to size of input
data

if length(interleaved) > K
% If length of input data is smaller than interleaver matrix, pad
zeros
pad_bits=length(interleaved)-K;
Input_data=[Input_data zeros(1,pad_bits)];
end        % end of if loop

% Interleaving the data
for cnt=1:length(interleaved)
    y(cnt)=Input_data(interleaved(cnt));
end     % end of for loop

y=y(1:K);   % removing padded bits from the interleaved data
end     %end of function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function to generate the interleaver matrix**

```
function [interleaved,R,U,C,T]=interleave(K)

% function for generating the interleaver matrix based on the size of
data
% input is size of the data K
% output is interleaver matrix interleaved
% number of rows of the matrix R
% number of columns of matrix C
% Factor determing the intra row permutation U
% Factor determining the inter row permutation T

% determine number of rows of the interleaver matrix

if  (K >= 40 ) && ( K<= 159)
    R=5;
```

```
elseif  ((K >= 160 ) && ( K<= 200) ) ||  ((K >= 481) && ( K<= 530) )
    R=10;
else
    R=20;
end     % end of if loop

% determine number to be used in intra permutation p and number of
columns C
prim_num=primes(260);
prim_tab=[3 2 2 3 2 5 2 3 2 6 3 5 2 2 2 2 7 5 3 2 3 5 2 5 2 6 3 3 2 3
2 2 6 5 2 5 2 2 2 19 5 2 3 2 3 2 6 3 7 7 6 3];
prim_num=prim_num(4:end);
cnt=1;

if  ((K >= 481) && ( K<= 530) )
    p=53;
    C=p;
    V=2;
else
     p=prim_num(cnt);
    while K > R *(p+1)
        cnt=cnt+1;
        p=prim_num(cnt);



    end % enf of while loop
  V=prim_tab(cnt);

if  K<=  R*(p-1)
    C=p-1;
elseif    ((R*(p-1)) < K ) && (K <=( R*p))
    C=p;
elseif    K > R*p
    C=p+1;
end  % end of inner ifelse loop

end     % end of outer if else loop

% generating array of data from 1 to size of matrix
x=1:(R*C);

% writing X in matrix form row wise
X=[];
cnt_r=0;
while cnt_r < R
    X=[X ; x((cnt_r*C)+1 :((cnt_r+1)*C))];
    cnt_r=cnt_r+1;
end     % end od while loop

% inter row and intrarow permutations
% step 1  primitive root v
% step 2  construct base sequence for intra row permutation
s=1;
for cnt_s=2:p-1
    s=[s mod((V*s(cnt_s-1)),p)];
end

% step 3 assign q0
```

```
q=1;   % qo=1
cnt_prim=1;
prim_gen=primes(1000);
prim_gen=prim_gen(4:end);
prim=prim_gen(cnt_prim);
cnt_q=2;
while cnt_q<=R
    mid_q=gcd(prim,p-1);
    if mid_q==1
        q(cnt_q)=prim;
        cnt_q=cnt_q+1;
    end
    cnt_prim=cnt_prim+1;
    prim=prim_gen(cnt_prim);
end

% step 4 permute q to get r
if  R==5
    T=fliplr([1:5]);
elseif  R==10
    T=fliplr([1:10]);
elseif (R==20) && (((2281 <= K) && (K <= 2480)) || ((3161 <= K) && (K
<= 3210)))
    T=[20 10 15 5 1 3 6 8 13 19 17 14 18 16 4 2 7 12 9 11];
else
    T=[20 10  15  5  1  3  6  8  13  19  11  9  14  18  4  2  17  7
16 12];
end

r=zeros(1,R);
for cnt_i=1:R
    r(T(cnt_i))=q(cnt_i);
end

% step 5 intra row permutation
U=zeros(R,C);
for cnt_i=1:R

    if C == p
        for cnt_j=0:p-2
            U(cnt_i,cnt_j+1)= s( (mod((cnt_j*(r(cnt_i))) ,(p-1) ))+1);

        end
    end

    if C== (p+1)
        for cnt_j=0:p-2
            U(cnt_i,cnt_j+1)= s(( mod((cnt_j*(r(cnt_i))) ,(p-1) ))+1);

        end

        U(cnt_i,p+1)=p;

        % exchanging first and last row
        if   K== (R*C)
            new_U=U;
            U(R,C)=new_U(R,1);
            U(R,1)=new_U(R,C);
```

```
        end
    end

    if C== (p-1)
        for cnt_j=0:p-2
            U(cnt_i,cnt_j+1)= (s(( mod((cnt_j*(r(cnt_i))) ,(p-1)
))+1))-1;

        end

    end

end

U=U+1;

% performing intra row permutation
intra_row=zeros(R,C);
for cnt_row=1:R
    for cnt_col=1:C
        intra_row(cnt_row,cnt_col)=X(cnt_row,U(cnt_row,cnt_col));

    end
end

% % (6) Perform the inter-row permutation for the rectangular matrix
based on the pattern ( ) i?{0,1, ,R?1}
inter_row=zeros(R,C);

for cnt=1:R
    inter_row(cnt,:)=intra_row(T(cnt),:);
end

Y=[];
for cnt_y=1:C
    Y=[Y (inter_row(:,cnt_y))'];
end
interleaved=Y;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function for bpsk modulation**

```
function [mod_signal]=bpsk_mod(turbo_encoded)
%bpsk modulate the data

mod_signal=zeros(1,length(turbo_encoded));
for cnt=1:length(turbo_encoded)
    if turbo_encoded(cnt)==0
    mod_signal(cnt)=-1;
    else
    mod_signal(cnt)=1;
    end
end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function to simulate channel models**

```
function [symbol_likelihood]= channel (
input,SNR,channel_type,symbols_per_frame,rate)

% converting from db to linear scale
EbNo =  10.^(SNR/10);
EsNo = rate.*EbNo;       % energy of symbol to noise
variance = 1/(2*EsNo);   % variance of noise
N0=2*variance;           % noise spectral density

% create the fading coefficients ,generate noise
if  (channel_type==1)  % AWGN channel
    a = ones(1,symbols_per_frame);

elseif (channel_type==2)  % Rayleigh fading
    a = sqrt(0.5)*( randn( 1,symbols_per_frame) + j*randn(
1,symbols_per_frame) );
   % noise =
sqrt(variance)*(randn(1,symbols_per_frame)+i*randn(1,symbols_per_frame
));
end

% add noise to the signal
noise = sqrt(variance)*(randn(1,symbols_per_frame));
r = abs(a).*input + noise;
symbol_likelihood = -2*r.*abs(a)/variance; % This is the LLR

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function to simulate the turbo decoder**

```
function
[tilda2_xk,xk_hat,err,w_xk]=turbo_dec(rx_signal,iter,data,w_xk,sterric
)
% main function for the decoder
% rx signal =received LLR
% iter=number of turbo code iterations to perform
% data = original data bits that will be used for calculating errors
% w_xk = feedback from 2nd decoder to 1st decoder

% the received signal rx_signal is in the form
% X1, Z1, Z1', X2, Z2, Z2', . . . , XK, ZK, ZK',XK+1, ZK+1, XK+2,
ZK+2, XK+3, ZK+3, XK'+1, ZK'+1, XK'+2, ZK'+2, XK'+3, ZK'+3,

length_data= length(rx_signal);      % checking length of the signal
received
data_bits=rx_signal(1:end-12);       % Seperating data bits from
received signal X1, Z1, Z1', X2, Z2, Z2', . . . , XK, ZK, ZK'
parity_bits=rx_signal(end-11:end);   % parity part   XK+1, ZK+1, XK+2,
ZK+2, XK+3, ZK+3, XK'+1, ZK'+1, XK'+2, ZK'+2, XK'+3, ZK'+3
K= (length_data-12)/3;               % length of the input signal

% separating data and encoded bits  Xk, Zk, Zk'
r_xk=data_bits(1:3:length_data-12);
r_zk=data_bits(2:3:length_data-12);
```

```
r_zk_dash=data_bits(3:3:length_data-12);

% appending flush bits Xk+1, Zk+1, Xk+2, Zk+2, Xk+3, Zk+3,
r_zk=[r_zk parity_bits(1:2:6)];
parity1=parity_bits(2:2:6);
% appending flush bits X'k+1,Z'k+1,X'k+2, Z'k+2, X'k+3, Z'k+3
r_zk_dash=[r_zk_dash parity_bits(7:2:12)];
parity2=parity_bits(8:2:end);
r_xk1=[r_xk parity1];
out=[];
% loop for iteration

for k_loop=1:iter
     % call for upper decoder
    v1_xk=r_xk1+w_xk;  % input 1
    r_zk;              % input 2
    [tilda1_xk]=dec_recursion(v1_xk,r_zk,sterric);    % output of 1st
decoder
    tilda1_xk=tilda1_xk(1:length(r_xk));  %  removing parity bits
before passing infor to 2nd decoder
    w_xk=w_xk(1:length(r_xk));            %
    v2_xk=tilda1_xk-w_xk;

    % Interleaving
    % dividing data into blocks for interleaving
    len_bits=length(r_xk);
    num_blocks=ceil(len_bits/5114);   % determining number of data
blocks
    data_in=v2_xk;                        % data to be interleaved
    interl_out=[];                        % interleaver output
    if  num_blocks>1
        for cnt_block=1:num_blocks-1
            input_interl=(data_in((((cnt_block-1) *5114)
+1):cnt_block*5114));
            interl_out=[interl_out interl(input_interl)];
        end
    end
        % inetrleaving the last block
    input_interl = data_in(  (    (   (num_blocks-1) *5114)  +1)
:end);
    [y]=interl(input_interl);
    interl_out=[interl_out y];     % all data blocks interleaved
       v2_xk_dash= [interl_out parity2];     % interleaved data +
parity

    % call for 2nd decoder
    [tilda2_xk_dash]=dec_recursion(v2_xk_dash,r_zk_dash,sterric);    %
output of 2nd decoder
    tilda2_xk_dash=tilda2_xk_dash(1:length(r_xk));

    % deinterleave the output
    % number of blocks=num_blocks
    deinterl_in=tilda2_xk_dash;            % data to be deinterleaved
    deinterl_out=[];                       % deinterleaver output
    if  num_blocks>1
        for cnt_block=1:num_blocks-1
            deinterl_in((((cnt_block-1) *5114) +1):cnt_block*5114);
            mid_dein=deinterl(deinterl_in((((cnt_block-1) *5114)
+1):cnt_block*5114));
            deinterl_out=[deinterl_out mid_dein]
```

```
        end
    end

    % inetrleaving the last block
    deinterl_last = deinterl_in(  (    (   (num_blocks-1) *5114)  +1)
:end);
    [y]=deinterl(deinterl_last);
    deinterl_out=[deinterl_out y];     % all data blocks deinterleaved
    tilda2_xk=deinterl_out;
    % updating w_xk for next iteration
    w_xk=tilda2_xk-v2_xk;
    w_xk=[w_xk zeros(1,3)];

end

tilda2_xk=tilda2_xk+tilda1_xk ;

% perform hard decision to check for errors
xk_hat=[];  % initializing output

for count_check=1:length(tilda2_xk)
    if tilda2_xk(count_check) > 0
        xk_hat(count_check)=1;
    elseif tilda2_xk <=0
        xk_hat(count_check)=0;
    end

end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function to deinterleaver the data**

```
function[output]=deinterl(xin)   % input data
% fucntion call for deinterleaver

K=length(xin);
[interleaved,R,U,C,T]=interleave(K); % function calll for calculating
the required parameters

% writing data in matrix form
size_matrix=R*C;

if K< size_matrix         % if data smaller than matrix size,padding
required
    xin=[xin (K+1):(size_matrix)];
end

% writing X in matrix form
X_out=[];
mid_in=xin;
for cnt_c=1:C
    X_out=[X_out (mid_in(1:R))'];
    mid_in=mid_in(R+1:end);
end
% X is now in matrix form
```

```
% reverse of inter row permutation
Inter_row=zeros(R,C);
for cnt=1:R
    Inter_row(T(cnt),:)=X_out(cnt,:);
end

%reverse of intra row permutation
Intra_row=zeros(R,C);
for cnt_row=1:R
    for cnt_col=1:C
        Intra_row(cnt_row,U(cnt_row,cnt_col)
)=Inter_row(cnt_row,cnt_col);
    end
end

% writing data in array
out_array=[];
for cnt_y=1:R
    out_array=[out_array (Intra_row(cnt_y,:))];
end

output=out_array(1:K);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **<u>Function to implement the constituent RSC decoder</u>**

```
function [LLR,LLR_coded] = dec_recursion(input1,input2,sterric)
% Function call for the component decoder
% Input 1 is the uncoded LLR
% Input 2 is the coded LLR
% Sterric defines the type of jacobian algorithm used

% The outputs from the code are:
% LLR= extrinsic LLR of the data bits
% LLR_c= estrinsic LLR of the coded bits

% The trellis structure is described by the trans_matrix.
% It gives the initial state,the final state, parity bit Z(i,j)
generated
% and the data bit X(i,j) input

    %            FromState,  ToState, ParityBit Z(i,j), DataBit
X(i,j)
   trans_matrix =  [1,          1,          0,          0;
                    2,          5,          0,          0;
                    3,          6,          0,          1;
                    4,          2,          0,          1;
                    5,          3,          0,          1;
                    6,          7,          0,          1;
                    7,          8,          0,          0;
                    8,          4,          0,          0;
                    1,          5,          1,          1;
                    2,          1,          1,          1;
                    3,          2,          1,          0;
                    4,          6,          1,          0;
```

```
                    5,          7,          1,          0;
                    6,          3,          1,          0;
                    7,          4,          1,          1;
                    8,          8,          1,          1];

    % count the number of total states
    state_cnt = max(trans_matrix(:,1));

    data_size=length(input1);                % size of the input data
    K=data_size-3;                           % data is from k=1...K+3
    num_transition= 2*state_cnt;             % number of transitions

    % calculating the state transition metrics or gammas
    % there are four distinct state transition metrics as marked on
the
    % trellsi diagram
    % gamma0 = 0
    % gamma1 = V(Xk) uncoded
    % gamma2 = R(Zk)  encoded
    % gamma3 = V(Xk)+R(Zk)

    % Two gamma matrices with V(Xk) and R(ZK) values and zeros

    gamma1=zeros(size(trans_matrix,1),data_size);  % matrix for V(Xk)
    gamma2=zeros(size(trans_matrix,1),data_size);  % matrix for R(Zk)
    for bit_cnt = 1:data_size
       for trans_cnt = 1:size(trans_matrix,1)
          if trans_matrix(trans_cnt, 3)==0          % when Z(i,j)=0
then gamma=X(i,j)V(Xk)
             gamma1(trans_cnt, bit_cnt) =input1(bit_cnt);
          end         % end of if
           if trans_matrix(trans_cnt, 4)==0         % when X(i,j)=0
then gamma=Z(i,j)R(Zk)
             gamma2(trans_cnt, bit_cnt) = input2(bit_cnt);
           end      % end of if
       end          % end of for for trans_cnt
       gamma=gamma1+gamma2;      % the state transition matrix gamma
?_ij=V(X_k )X(i,j)+R(Z_k )Z(i,j)
    end         % end of bit_cnt for loop

    %            Backward recursion

    betas=zeros(state_cnt,data_size);
    betas=betas-inf;                    % initializing beta to -  Inf
    betas(1,data_size)=0;               %  B(k+3)(S0)=0

    % starting from k= K+2 and going to k=1

    for bit_cnt = data_size-1:-1:1
        if (bit_cnt<=K)
            app_in = input1(bit_cnt);
        else
            app_in = 0;
        end

        beta_mid=zeros(state_cnt,1);
        for trans_cnt = 1:state_cnt
            beta1=betas(trans_matrix(trans_cnt,2),bit_cnt+1)
+gamma(trans_cnt, bit_cnt+1);     % zero state connected to state S
```

```
            beta2=betas(trans_matrix(trans_cnt+8,2),bit_cnt+1)+
gamma(trans_cnt+8, bit_cnt+1); % one state connected to state S

beta_mid(trans_matrix(trans_cnt,1))=max_sterric(beta1,beta2,sterric);
% max_sterric of both the betas
             betas(trans_matrix(trans_cnt,1),bit_cnt) =
max_sterric(beta1,beta2,sterric);
        end         % end of for loop for trans_cnt
     beta_mid=beta_mid-((ones(state_cnt,1))*(beta_mid(1,1)));
nomalizing Beta(Si)-Beta(S0)
        betas(:,bit_cnt)=beta_mid;                                    %
saving values in beta matrix

    end             % end of for loop for bit_cnt

    %               Forward recursion

    alphas=zeros(state_cnt,data_size);
    alphas=alphas-inf;
    alphas(1,1)=0;                  % alpha0(so)=0
    for bit_cnt = 2:data_size       % statring from state 2 and going
till the end
        if bit_cnt <= K
            app_in=input1(bit_cnt-1);
        else
            app_in=0;
        end

       for trans_cnt = 1:num_transition
            if alphas(trans_matrix(trans_cnt,1),bit_cnt-1) ==0
                app_in=0;
            end
            alpha1=alphas(trans_matrix(trans_cnt,1),bit_cnt-1)
+gamma(trans_cnt, bit_cnt-1);
            %sterric=1;

alphas(trans_matrix(trans_cnt,2),bit_cnt)=max_sterric(alpha1,alphas(tr
ans_matrix(trans_cnt,2),bit_cnt),sterric);

       end      % end of for loop for trans_cnt
        alpha_mid=alphas(:,bit_cnt);  % normalizing alphas
         alpha0=alpha_mid(1,1);
         alpha_mid=alpha_mid-alpha0;
         alphas(:,bit_cnt)=alpha_mid  ;
    end         % end of for loop for the bit_cnt


     % LLR estimate for data bit Xk.
    lamdas=zeros(num_transition,data_size);
    for bit_cnt = 1:data_size
       for trans_cnt = 1:size(trans_matrix,1)
           lamdas(trans_cnt, bit_cnt) =
alphas(trans_matrix(trans_cnt,1),bit_cnt) + gamma(trans_cnt,bit_cnt) +
betas(trans_matrix(trans_cnt,2),bit_cnt);
       end
    end

    % Calculating likelihood of data 1
    LLR = zeros(1,data_size);
    for bit_cnt = 1:data_size
```

```
        % initializing the probability variables of -Inf
       lamda_0=-inf;
       lamda_1=-inf;
       for trans_cnt = 1:size(trans_matrix,1)
           lamda_mid=lamdas(trans_cnt,bit_cnt);
           if trans_matrix(trans_cnt,3)==0   % Z(i,j)=0 ----->
?_ij=V(X_k )X(i,j)
               lamda_0 = max_sterric(lamda_0,lamda_mid ,sterric);   %
max* of all lamdas for Xi=0
           else
              lamda_1 = max_sterric(lamda_1, lamda_mid,sterric);    %
max* of all lamdas for Xi=1
           end
       end
       LLR(bit_cnt) = lamda_0-lamda_1;         % LLR of bit being zero
or one
    end


    % extrinsic LLR estimate for coded bits
     % LLR estimate for code bits.
    lamdas2=zeros(num_transition,data_size);
    for bit_cnt = 1:data_size
       for trans_cnt = 1:size(trans_matrix,1)
           lamdas2(trans_cnt, bit_cnt) =
alphas(trans_matrix(trans_cnt,1),bit_cnt) + gamma2(trans_cnt,bit_cnt)
+ betas(trans_matrix(trans_cnt,2),bit_cnt);
       end
    end

    % Calculating likelihood of data 1
    LLR_coded = zeros(1,data_size);
    for bit_cnt = 1:data_size
        % initializing the probability variables of -Inf
       lamda_0c=-inf;
       lamda_1c=-inf;
       for trans_cnt = 1:size(trans_matrix,1)
           lamda_mid2=lamdas2(trans_cnt,bit_cnt);
           if trans_matrix(trans_cnt,4)==0   % Z(i,j)=0 ----->
?_ij=V(X_k )X(i,j)
               lamda_0c = max_sterric(lamda_0c,lamda_mid2 ,sterric);
% max* of all lamdas for Xi=0
           else
              lamda_1 = max_sterric(lamda_1c, lamda_mid2,sterric);
% max* of all lamdas for Xi=1
           end
       end
       LLR_coded(bit_cnt) = lamda_0c-lamda_1c;         % LLR of bit
being zero or one
    end

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function to compute the max* algorithm**

```
function out=max_sterric(x,y,sterric)
% function for calculaing the Jacobian algorithm
if (x== -Inf && y== -Inf)
    out= -Inf;
elseif sterric==1   % compute log MAP
    ab=abs(x-y);
    fc=(log(1+exp(-ab)))/log(2);
    max_s=max(x,y)+fc;
    [out]=max_s;

elseif sterric==2   %comput max-log MAP
    max_s=max(x,y);

elseif sterric==3   % comput constant-log MAP
    ab=abs(y-x);
    T=1.5;          % values as specified in text
    C=0.5;
    if ab > T
        fc=0;
    elseif ab <= T
        fc=C ;
    end
    max_s=max(x,y)+fc;
    out=max_s;

elseif sterric==4   % compute linear-log MAP
    ab=abs(y-x);
    T=2.5068;       % constant values as specified in text
    alpha=-0.24904;
    if ab > T
        fc=0;
    elseif ab <= T
        fc=alpha*(ab-T) ;
    end
    max_s=max(x,y)+fc;
    out=max_s;
end

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function for hard decision decoding**

```
function [xk_hat]=hard_decision(tilda2_xk)
% input is the LLR sequenct tilda2_xk
% output is the hard decoded bit sequence xk_hat

xk_hat=[];  % initializing output

for count_check=1:length(tilda2_xk)
    if tilda2_xk(count_check) > 0
        xk_hat(count_check)=1;
    elseif tilda2_xk <=0
        xk_hat(count_check)=0;
```

```
    end
end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function for calculating average mutual information**

```
% llrs is a 1xK vector of LLRs
% mutual_information is a scalar in the range 0 to 1
function mutual_information = avg_mutual_inf(llrs)

    P0 = exp(llrs)./(1+exp(llrs));
    P1 = 1-P0;
    entropies = -P0.*log2(P0)-P1.*log2(P1);
    mutual_information = 1-
sum(entropies(~isnan(entropies)))/length(entropies);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Main function for plotting the EXIT chart**

```
% The main Program for generating exit chart for the UMTS Turbo code
clear all
close all
clc

K=40;                           % Specify frame size between 40 and
5114 bits.
SNR =0;                         % Choose the SNR
frame_count_max = 100;          % Choose how many frames to simulate
IA_count = 10;                  % Choose how many points to plot in the
EXIT functions
channel_type=1;                 % AWGN channel
sterric=1;
symbols_per_frame=K*3+12;         % Symbols in frame after encoding.
rate=K/symbols_per_frame;         % Code rate of the Turbo code (ratio
of number of input bits to output bits)

% Calculate the MIs to use for the a priori LLRs
IAs = (0:(IA_count-1))/(IA_count-1);
IE_means = zeros(1,IA_count);
IE_stds = zeros(1,IA_count);
% Determine each point in the EXIT functions

for IA_index = 1:IA_count       % loop for IA count
    IEs = zeros(1,frame_count_max);     % initializng IE to all zeros

    % This runs the simulation long enough to produce smooth EXIT
functions.
    for frame_index = 1:frame_count_max

        data=randint(1,K);    % generate random data
```

```
        % function call for turbo code
        [turbo_encoded]=turbo_enc(data);

        %function call to bpsk modulate the turbo coded signal
        [mod_signal]=bpsk_mod(turbo_encoded);

        % Function call to pass through the channel, demodulate and
        % generate LLRs

        [symbol_likelihood]= channel ( mod_signal , SNR
,channel_type,symbols_per_frame,rate);
        w_xk=generate_llrs(data, IAs(IA_index));    % generate random
LLRs to use as reference
        rx_signal=symbol_likelihood;
        data_bits=rx_signal(1:end-12) ;      % data part
        parity_bits=rx_signal(end-11:end);   % parity part   XK+1,
ZK+1, XK+2, ZK+2, XK+3, ZK+3, XK'+1, ZK'+1, XK'+2, ZK'+2, XK'+3,
ZK'+3,
        length_data=length(data_bits);

        % seperating the systematic bits and coded bits
        r_xk=data_bits(1:3:length_data);
        r_zk=data_bits(2:3:length_data);
        r_zk_dash=data_bits(3:3:length_data);

        % appending flush bits Xk+1, Zk+1, Xk+2, Zk+2, Xk+3, Zk+3,
        % X'k+1,Z'k+1,X'k+2, Z'k+2, X'k+3, Z'k+3
        r_zk=[r_zk parity_bits(1:2:6)];
        parity1=parity_bits(2:2:6);
        r_zk_dash=[r_zk_dash parity_bits(7:2:12)];
        parity2=parity_bits(8:2:end);
        r_xk1=[r_xk parity1];

        % loop for iteration
        w_xk=[w_xk 0 0 0];   % extrinsic information
        v1_xk=r_xk1+w_xk;    % apriori information

        [tilda1_xk]=dec_recursion(v1_xk,r_zk,sterric);          %
function call for RSC decoding
        IEs(frame_index) = avg_mutual_inf(tilda1_xk);           %
calculate the mutual info of the decoder output.This is the extrinsic
information.

    end

    % Store the mean and standard deviation of the results
    IE_means(IA_index) = mean(IEs);
    IE_stds(IA_index) = std(IEs);
end

figure;
axis square;
title('EXIT chart for BPSK modulation in an AWGN channel');
ylabel('I_E');
xlabel('I_A');
xlim([0,1]);
ylim([0,1]);

hold on;
```

```
% Plot the EXIT function for component decoder 1
plot(IAs,IE_means,'-');
plot(IAs,IE_means+IE_stds,'--');
plot(IAs,IE_means-IE_stds,'--');

% Plot the inverted EXIT function for component decoder 2
plot(IE_means,IAs,'-');
plot(IE_means+IE_stds,IAs,'--');
plot(IE_means-IE_stds,IAs,'--');

% saving data in mat file
filename =
['AWGN_',num2str(K),'_','logMAP_','exit_chart_',num2str(SNR),'.mat'];
save(filename, 'IAs','IE_means', '-MAT');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- **Function call for the decoder trajectory in EXIT chart**

```
% The main Program for generating the UMTS Turbo code in MATLAB
clear all
close all
clc
figure
K=320;                              % Specify frame size between 40
and 5114 bits.
iter_cnt = 20;                      % Specify the maximum number of
decoding iterations to perform.
symbols_per_frame=K*3+12;           % Symbols in frame after encoding.
rate=K/symbols_per_frame;           % Code rate of the Turbo code
(ratio of number of input bits to output bits)
frame_count_max=10;                 % Maximum number of frames to be
passed to the code
bit_count_max=frame_count_max*K;    % Maximum bits at a given SNR
error_max=10;                       % Maximum number of errors (to be
used in condition later)
count_SNR=0;                        % Counter for number of SNR values
results=[];                         % The ouput results matrix defined
SNR=0;
channel_type=1;                     % channel_type can be 1 or 2(for
AWGN and Rayleigh fading respectively)
sterric=1  ;                        % sterric can be 1 to 4 (for
LogMAP, max-logMAP, constant-logMAP, linear-logMAP)
max_fail_count = 3;                 % Specift the number of iterations
that should fail to improve the decoding before the iterations are
stopped early

% calculating Lc
EbNo =  10.^(SNR/10);
EsNo = rate.*EbNo;                  % energy of symbol to noise
variance = 1/(2*EsNo);              % variance of noise
Lc=2/variance;

% Initiazing the output arrays
IA1mean=zeros(1,iter_cnt);
IE1mean=zeros(1,iter_cnt);
IA2mean=zeros(1,iter_cnt);
IE2mean=zeros(1,iter_cnt);
```

```
figure;
axis square;
title('BPSK modulation in an AWGN channel');
ylabel('I_E');
xlabel('I_A');
xlim([0,1]);
ylim([0,1]);
hold on;

for frame_index = 1:frame_count_max
    err_out=[];                         % error count matrix
    BER=1;                              % initializig BER to max i.e. 1
    data=randint(1,K);                  % generate random data
    results=[zeros(1,2)];
    error_counts=zeros(1,iter_cnt);  % initialize error count matrix
    bit_count=0;

    % function call for turbo code
    [turbo_encoded]=turbo_enc(data);

    %function call to bpsk modulate the turbo coded signal
    [mod_signal]=bpsk_mod(turbo_encoded);

    % Function call to pass through the channel

    [symbol_likelihood]= channel ( mod_signal , SNR
,channel_type,symbols_per_frame,rate);
    extrinsic2=zeros(1,K);    % feedback from second copmponent
encoder to first component encoder.It is initialized to zero before
first iteration.

    avg_IA = 1;               % initializing the mutual information
averaging value to zero
    iteration_index = 1;      % iteration index is initialized to 1
    fail_count = 0;           % number of times the code has failed
    IA1 = 0;
    IE1 = 0;
    IA2=0;
    IE2=0;

    % Run the simulation while the number of failed iterations
    % is less than max_fail_count and iteration index is less than
    % maximum ietartions

    % seperating the data and parity bits from the received signal
    rx_signal=symbol_likelihood;
    length_data= length(rx_signal);      % checking length of the
signal received
    data_bits=rx_signal(1:end-12);       % data part
    parity_bits=rx_signal(end-11:end);   % parity part   XK+1, ZK+1,
XK+2, ZK+2, XK+3, ZK+3, XK'+1, ZK'+1, XK'+2, ZK'+2, XK'+3, ZK'+3,
    K= (length_data-12)/3;             % length of the input signal


    % separating data and encoded bits  Xk, Zk, Zk'
    r_xk=data_bits(1:3:length_data-12);
    r_zk=data_bits(2:3:length_data-12);
    r_zk_dash=data_bits(3:3:length_data-12);
```

```
    % appending flush bits Xk+1, Zk+1, Xk+2, Zk+2, Xk+3, Zk+3,
    % X'k+1,Z'k+1,X'k+2, Z'k+2, X'k+3, Z'k+3

    r_zk=[r_zk parity_bits(1:2:6)];
    parity1=parity_bits(2:2:6);
    r_zk_dash=[r_zk_dash parity_bits(7:2:12)];
    parity2=parity_bits(8:2:end);


    while  iteration_index <= iter_cnt

        w_xk=extrinsic2;        % extrinsic information 2
        apriori1=extrinsic2;    % apriori information 1
        IA1 = avg_mutual_inf(apriori1);
        % function call to turbo decoder. Ouputs are extrinsic
information
        % for decoder 1 and 2 and apriori information for decoder 2.

[extrinsic1,extrinsic2,apriori2]=turbo_dec_traj(r_xk,r_zk,r_zk_dash,Lc
, K, apriori1,sterric);

        % calculate IAs and IEs
        IE1 = avg_mutual_inf(extrinsic1);
        IA2 = avg_mutual_inf(apriori2);
        IE2 = avg_mutual_inf(extrinsic2);

        % store the results in output arrays
        IA1mean(1,iteration_index)=IA1mean(1,iteration_index)+IA1;
        IE1mean(1,iteration_index)=IE1mean(1,iteration_index)+IE1;
        IA2mean(1,iteration_index)=IA2mean(1,iteration_index)+IA2;
        IE2mean(1,iteration_index)=IE2mean(1,iteration_index)+IE2;

        iteration_index = iteration_index + 1;   % increment the
iteration index

    end       % end of while loop for iterations

end         % end of for loop for frame count

% calculating means of IAs and IEs

IA1mean=(IA1mean)./frame_count_max
IE1mean=(IE1mean)./frame_count_max
IA2mean=(IA2mean)./frame_count_max
IE2mean=(IE2mean)./frame_count_max

% plotting the trjectory
j=0;
figure
for i=1:length(IA1mean)
    j=j+1;
    traj(j)=IE1mean(i);
    indices(j)=IA1mean(i);
    j=j+1;
    traj(j)=IA2mean(i);
    indices(j)=IE2mean(i);
end
plot(indices,traj,'k.-')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```